YRT-PET

Yale PET Center

May 19, 2026

COMPILATION

1 Building YRT-PET **1**
- 1.1 Requirements . . . 1
- 1.2 Configuration and compilation . . . 1
- 1.3 FAQ . . . 2

2 Adding plugins to YRT-PET **5**

3 Command line interface **7**

4 Python interface **9**

5 YRT-PET Configuration **11**
- 5.1 Number of threads . . . 11
- 5.2 Disabling page-locked memory (or pinned memory) . . . 11
- 5.3 From Python . . . 11

6 Dynamic Framing **13**
- 6.1 Python Usage . . . 13
- 6.2 File Format . . . 15
- 6.3 Notes . . . 15

7 Data formats **17**
- 7.1 Scanner (`Scanner`) . . . 17
- 7.2 Image format (`Image`) . . . 17
- 7.3 YRT-PET raw data format (`Array`) . . . 17
- 7.4 Listmode (`ListmodeLUT`) . . . 17
- 7.5 Sparse histogram (`SparseHistogram`) . . . 17
- 7.6 Motion information (`LORMotion`) . . . 17
- 7.7 Histogram (`Histogram3D`) . . . 18

8 Scanner definition **19**
- 8.1 Scanner parameters . . . 19
- 8.2 Look-Up-Table . . . 20

9 Image **23**
- 9.1 Image parameters . . . 23
- 9.2 Image file format . . . 24
- 9.3 Fourth Dimension (Time) . . . 24
- 9.4 For Python users and plugin developers . . . 24

10 List-mode file **25**

10.1 For Python users . . . 25

11 List-mode DOI file **27**

12 Motion information file **29**

13 Histogram3D format **31**
13.1 File format . . . 31
13.2 Array format . . . 31
13.3 Crystals in the same ring . . . 32
13.4 Binning for different rings . . . 33
13.5 Dealing with multiple DOI layers . . . 34
13.6 Histogram Dimensions . . . 35
13.7 Example . . . 36
13.8 Small asymmetry in the Histogram . . . 36

14 RAWD format **39**

15 Sparse histogram file **41**
15.1 Implementation . . . 41

16 Image-Based PSF File Format **43**
16.1 Uniform PSF Kernel . . . 43
16.2 Spatially Variant PSF Kernel (PSF LUT) . . . 44

17 Array **47**
17.1 Overview . . . 47
17.2 Supported Types . . . 47
17.3 Memory management . . . 47
17.4 Data Access . . . 48
17.5 NumPy Interoperability . . . 51
17.6 Boolean Array example . . . 51

18 Vector3D and Line3D **53**
18.1 Vector3D . . . 53
18.2 Line3D . . . 56
18.3 Practical Example: Creating LORs for Projection . . . 58
18.4 Notes . . . 59

19 Projector **61**
19.1 Available Projector Types . . . 61
19.2 Python Usage . . . 61
19.3 Siddon vs Distance-Driven . . . 63
19.4 Siddon-specific Methods . . . 63
19.5 Single LOR Projection Methods (Siddon) . . . 63
19.6 Notes . . . 64

20 Bin Iterator **65**
20.1 Python Types . . . 65
20.2 Available Iterators . . . 65
20.3 For plugin developers . . . 65

21 OperatorProjector **67**
21.1 Python Usage . . . 67
21.2 Important methods . . . 70

22 ProjectionList **71**
22.1 Python Usage 71
22.2 Owned vs Alias 72

23 Owned vs Alias (For Python users and Plugin developers) **73**
23.1 Examples 73

24 OSEM **77**
24.1 Basic Usage 77
24.2 Configuration 78
24.3 Output 79

25 Constraints **81**
25.1 Python Types 81

26 Frequenty asked questions **83**

27 Contributing **85**
27.1 Git feature development workflow 85
27.2 Testing 85
27.3 Code standard 86
27.4 Reconstruction code basis 86
27.5 Priority functionality 86
27.6 Wish list for a full product 87

28 Current YRT-PET version: **89**

29 Citing YRT-PET: **91**

CHAPTER

ONE

BUILDING YRT-PET

1.1 Requirements

- `pybind11` if compiling the python bindings (ON by default)
- CUDA toolkit if compiling using GPU (ON by default)
- An internet connection to download the `cxxopts`, `nlohmann/json`, and `catch2` libraries
- zlib, to read NIfTI images in `.nii.gz` format, but this is pre-installed in most Unix distributions

1.2 Configuration and compilation

From the command-line interface:

```
git clone git@github.com:YaleBioImaging/yrt-pet.git
cd yrt-pet
mkdir build
cd build
cmake ../yrt-pet/ -DUSE_CUDA=[ON/OFF] -DBUILD_PYBIND11=[ON/OFF]
make
```

With `[ON/OFF]` being replaced by the desired configuration

- The `-DUSE_CUDA` option enables or disables GPU accelerated code
  - This option is `ON` by default
- The `-DBUILD_PYBIND11` option enables or disables YRT-PET's python bindings
  - This option is `ON` by default

1.2.1 Post-compilation steps

- (optional) To run unit tests, run `ctest -V` from the build folder.
- Install YRT-PET by running `cmake --install . --prefix <installation path>` from the build folder.
  - This will install YRT-PET's header files, library, and binary executables in `<installation path>`
  - A file named `install_manifest.txt` will be created in the build folder
  - To uninstall, run `xargs rm < ./install_manifest.txt`
- To check if GPU was successfully enabled for the project, run `yrtpet_reconstruct --help`. If the `--gpu` option appears, the program was compiled with GPU acceleration.

1.3 FAQ

- I get a message that look like `Could NOT find ZLIB (missing:  ZLIB_LIBRARY ZLIB_INCLUDE_DIR)`
  - This means that the zlib library was not found
  - Remedy for Linux:
    - if you use APT: `sudo apt install zlib1g-dev`
    - if you use YUM: `sudo yum install zlib-devel`
  - Remedy for macOS:
    - `brew install zlib-devel`
  - It is a widely used library and is widely available on many platforms. Make sure to check online if the above solutions do not work.
- I get a message that looks like:

```
Could not find a package configuration file provided by "pybind11" with any
of the following names:

  pybind11Config.cmake
  pybind11-config.cmake

Add the installation prefix of "pybind11" to CMAKE_PREFIX_PATH or set
"pybind11_DIR" to a directory containing one of the above files.  If
"pybind11" provides a separate development package or SDK, be sure it has
been installed.
```

  - This is because the `pybind11` library was not found. YRT-PET requires the `pybind11` sources and CMake files to compile with python bindings. Several fixes are possible:
    - Disable Python bindings altogether by adding `-DBUILD_PYBIND11=OFF` to the CMake command
    - If you are using Linux with APT: `sudo apt install pybind11-dev`
    - On macOS: `brew install pybind11`
    - Another fix is to install `pybind11` using `pip` or `conda`. (See documentation for more information)
- If compiling with GPU acceleration enabled, note that by default, the architecture that the code will be compiled towards will be `native`. This means that YRT-PET will be compiled for the architecture of the host's GPU. Note that YRT-PET requires CMake 3.28+ to build the project
  - One can bypass this behavior using one of the two following ways:
    - Set the `CUDAARCHS` environment variable.
    - Add `-DCMAKE_CUDA_ARCHITECTURES=[CUDA architectures list]`.
  - Example: `-DCMAKE_CUDA_ARCHITECTURES="61;75"`
  - More information here
- If compiling with GPU acceleration enabled, make sure the `CUDACXX` environment variable is set to the location of `nvcc`. Run `echo $CUDACXX` to verify this.
- In order to compile with the python bindings, one needs to have a working python installation or activate a virtual environment.

  - The python bindings will only work for the host's Python version and only for the CPython implementations.
    * Example: One cannot compile YRT-PET with python bindings using Python 3.10 and expect them to work within Python 3.11
  - To add the YRT-PET python bindings to the python environment, add the `build` folder to the `PYTHONPATH` environment variable.
  - To test the python bindings, run: `python -c "import pyyrtpet as yrt; print(yrt.compiledWithCuda());"`
    * If the `PYTHONPATH` environment variable is not set or misplaced, the error will look like:
      · `ModuleNotFoundError: No module named 'pyyrtpet'`
    * If there is a mismatch between the python version used to compile YRT-PET and the environment's python version, the error will look like:
      · `ImportError: No module named pyyrtpet_wrapper._pyyrtpet`
    * If the `PYTHONPATH` environment is properly set and the python versions match, the command will either print `True` or `False`.
      · `True` if the project was compiled with `-DUSE_CUDA=ON`
      · `False` if the project was compiled with `-DUSE_CUDA=OFF`

- YRT-PET uses `std::thread` for parallelization. By default, YRT-PET will use the maximum amount of available threads.
- I get an error that looks like:

```
CMake Error at /usr/share/cmake-3.29/Modules/CMakeDetermineCompilerId.cmake:814
(message):
Compiling the CUDA compiler identification source file
"CMakeCUDACompilerId.cu" failed.

...
143 | #error -- unsupported GNU version! gcc versions later than 12 are not
supported! The nvcc flag '-allow-unsupported-compiler' can be used to override this
version check; however, using an unsupported host compiler may cause compilation
failure or incorrect run time execution. Use at your own risk.
|       ^~~~~
```

  - This is because `nvcc` tries to use a version of `gcc` that is not supported by the CUDA toolkit (yet).
  - If you have a different version of `gcc` installed, you can do: `-DCMAKE_CUDA_HOST_COMPILER=g++-11`

CHAPTER

TWO

ADDING PLUGINS TO YRT-PET

YRT-PET can be compiled alongside external plugins that can add several things:

- Additional List-mode or Histogram data format support
- Additional executables added to the compilation pipeline
- Additional Python interface functions

Plugins take the form of a folder with source code and a CMakeLists.txt file. This folder has to be copied (or symbolically linked in) the `plugins` directory of this repository.

See the `yrt-pet-csv` repository for a minimal example of a plugin that adds support for a CSV list-mode format.

CHAPTER

THREE

COMMAND LINE INTERFACE

The compilation directory should contain a folder named `executables`. The following executables might be of interest:

- `yrtpet_reconstruct`: Reconstruction executable for OSEM. Includes sensitivity image generation
- `yrtpet_forward_project`: Forward project an image into a fully 3D histogram
- `yrtpet_backproject`: Backproject a list-mode or a histogram into an image
- `yrtpet_convert_to_histogram`: Convert a list-mode (or any other datatype input) into a fully 3D histogram or a sparse histogram
- (Subject to change) `yrtpet_estimate_scatter`: Prepare a fully 3D histogram for usage in OSEM as scatter estimate. Currently experimental and incomplete

CHAPTER

FOUR

PYTHON INTERFACE

If the project is compiled with BUILD_PYBIND11, the compilation directory should contain a folder named pyyrtpet. To use the python library, add the compilation directory to your PYTHONPATH environment variable:

```
export PYTHONPATH=${PYTHONPATH}:<compilation folder>
```

Almost all the functions defined in the header files have a Python bindings. more thorough documentation on the python library is still to be written.

CHAPTER

FIVE

YRT-PET CONFIGURATION

5.1 Number of threads

Since YRT-PET currently uses the `std::thread` library to parallelize work, the thread selection is managed by that library. YRT-PET uses the maximum number of available threads unless the `--num_threads` is passed to the executables.

Alternatively, one can run YRT-PET (or any process) using `taskset` to limit CPU core selection.

5.1.1 From Python

Using the Python bindings, it is possible to call `yrt.setNumThreads(...)` to set the number of threads that will used for parallelized operations. This will only affect the current process.

The `yrt.getNumThreads()` function also exists to gather that information.

5.2 Disabling page-locked memory (or pinned memory)

For GPU operations, the intermediary buffers are allocated as page-locked memory. This increases speed as it can allow for asynchronous copies between host and device.

It is possible, however, to disable this behavior by setting the `YRTPET_DISABLE_PINNED_MEMORY` environment variable to `yes`.

5.3 From Python

Using the Python bindings, it is possible to call `yrt.setPinnedMemoryEnabled(...)` to define this option. This will not alter any environment variable, it will only affect the current process.

The `yrt.isPinnedMemoryEnabled()` function also exists to gather that information.

CHAPTER

SIX

DYNAMIC FRAMING

Dynamic framing manages timestamp ranges for dynamic reconstructions.

The framing defines the time ranges used for assigning a dynamic frame to each PET list-mode event. This is used for dynamic reconstruction as well as simple forward-backward projection operations.

This is used along the fourth dimension of the Image class, which only defines the *number* of dynamic frames, not the time range associated to each one.

The `DynamicFraming` object is used by the `OSEM` or the `OperatorProjector` to correctly map each list-mode event to the appropriate frame in the fourth dimension of the image space, allowing 4D reconstruction or projection.

The timestamps provided are in milliseconds and are in the same referrential as the timestamps of the list-mode or the motion file.

6.1 Python Usage

```
import pyyrtpet as yrt
import numpy as np

# ============================================================================
# Method 1: Create dynamic framing with a specified number of frames
# ============================================================================

# Create a DynamicFraming with 10 frames (requires setting timestamps afterwards)
framing = yrt.DynamicFraming(num_frames=10)

# Set frame start times - timestamps must be in chronological order
# Frame 0 starts at 0 ms
framing.setStartingTimestamp(frame=0, timestamp=0)
# Frame 1 starts at 10 ms
framing.setStartingTimestamp(frame=1, timestamp=10)
# Frame 2 starts at 20 ms
framing.setStartingTimestamp(frame=2, timestamp=20)
# Continue for all frames...
framing.setStartingTimestamp(frame=3, timestamp=30)
framing.setStartingTimestamp(frame=4, timestamp=45)
framing.setStartingTimestamp(frame=5, timestamp=60)
framing.setStartingTimestamp(frame=6, timestamp=80)
framing.setStartingTimestamp(frame=7, timestamp=100)
framing.setStartingTimestamp(frame=8, timestamp=125)
framing.setStartingTimestamp(frame=9, timestamp=135)
```

(continues on next page)

(continued from previous page)

```
# The last timestamp marks the end of the last frame
framing.setLastTimestamp(timestamp=150)

# Verify the framing is valid (timestamps in chronological order)
# This will also fail if some frames were left unset.
assert framing.isValid(), "Timestamps must be in chronological order"

# Get the number of frames
num_frames = framing.getNumFrames()
assert num_frames == 10, f"Expected 10 frames, got {num_frames}"

# ============================================================================
# Method 2: Create dynamic framing from a numpy array of timestamps
# ============================================================================

# Frame timestamps: start of each frame + end timestamp
# This creates 3 frames: [0-10ms], [10-30ms], [30-60ms]
# The dtype of the numpy array must be `uint32`
timestamps = np.array([0, 10, 30, 60], dtype=np.uint32)
framing_from_array = yrt.DynamicFraming(frame_timestamps=timestamps)
assert framing_from_array.getNumFrames() == 3
# Note that the array has 4 elements, but we defined 3 frames.
# This is because the last timestamp defines the end of the last frame.

# ============================================================================
# Method 3: Load dynamic framing from a file (.dyn extension)
# ============================================================================

# For this documentation's purposes only, we use a tempfile
import tempfile
with tempfile.NamedTemporaryFile(suffix='.dyn', delete=False) as f:
    my_file = f.name

# Framing data is stored as a text file containing timestamps separated by a
#  whitespace

# To save:
framing.writeToFile(my_file)
# To load:
framing_from_file = yrt.DynamicFraming(my_file)

# Check integrity
assert framing_from_file.getNumFrames() == framing.getNumFrames()
for i in range(framing.getNumFrames()):
    assert framing_from_file.getStartingTimestamp(i) == framing.getStartingTimestamp(i)
assert framing_from_file.getLastTimestamp() == framing.getLastTimestamp()

# ============================================================================
# Query methods
# ============================================================================

# Get total duration of the dynamic framing (last timestamp - first timestamp)
```

(continues on next page)

(continued from previous page)

```
total_duration = framing.getTotalDuration()
assert total_duration == 150, f"Expected 150ms, got {total_duration}"

# Get duration of a specific frame
frame_duration = framing.getDuration(frame=0)  # Duration of frame 0 (10 ms)
assert frame_duration == 10, f"Expected 10ms, got {frame_duration}"

# Get the timestamp when a specific frame starts
start_ts = framing.getStartingTimestamp(frame=0)  # 0
start_ts = framing.getStartingTimestamp(frame=5)  # 60

# Get the timestamp when a specific frame ends (i.e., next frame starts)
stop_ts = framing.getStoppingTimestamp(frame=0)  # 10 (start of frame 1)
stop_ts = framing.getStoppingTimestamp(frame=4)  # 80 (start of frame 5)

# Get number of timestamps (frames + 1 for the final timestamp)
num_timestamps = framing.getNumTimestamps()  # 11 (10 frames + 1 end)
```

6.2 File Format

The dynamic framing can be saved to a text file with the `.dyn` extension. Each line contains a single timestamp (in ms):

```
0
10
30
60
120
180
```

The file must contain at least 2 timestamps (defining 1 frame).

6.3 Notes

- Timestamps must be strictly increasing (each frame must start after the previous, without overlap)
- The number of timestamps equals `num_frames + 1` (frame starts + final timestamp)

CHAPTER

SEVEN

DATA FORMATS

Note that all binary formats encode numerical values in little endian.

7.1 Scanner (`Scanner`)

Scanners are defined in two parts:

- A scanner parameters file in JSON format
- A Look-Up-Table (LUT) file in binary format.

See *Documentation on the scanner definition* for more details

7.2 Image format (`Image`)

Images are read and stored in NIfTI format. YRT-PET also uses a JSON file to define the Image parameters (size, voxel size, offset, time dimension). See *Documentation on the Image parameters format*.

7.3 YRT-PET raw data format (`Array`)

YRT-PET stores its array structures in the RAWD format. See *Documentation on the RAWD file structure*

7.4 Listmode (`ListmodeLUT`)

YRT-PET defines a generic default List-Mode format. When used as input, the format name is `LM`. See *Documentation on the List-Mode file*

7.5 Sparse histogram (`SparseHistogram`)

YRT-PET defines a generic default sparse histogram format. When used as input, the format name is `SH`. See *Documentation on the sparse histogram file*

7.6 Motion information (`LORMotion`)

Motion information is encoded in a binary file describing the transformation of each frame. See *Documentation on the Motion information file*

7.7 Histogram (`Histogram3D`)

Fully 3D Histograms are stored in YRT-PET's RAWD format *described earlier*. Values are encoded in `float32`. The histogram's dimensions are defined by the scanner properties, which are defined in the `json` file *described earlier*.

See *Documentation on the histogram format* for more information. When used as input, the format name is `H`.

CHAPTER EIGHT

SCANNER DEFINITION

In YRT-PET, a scanner is defined in two parts: the scanner parameters file (JSON format) and the Look-Up-Table (LUT) file in binary format.

8.1 Scanner parameters

The following parameters in the JSON file define the scanner:

- `VERSION` : Scanner file format version. The current version is `3.2`. **[Mandatory]**
- `scannerName` : Scanner name. This is value is not used in the reconstruction. **[Mandatory]**
- `detCoord`: Path to the LUT file. The path is relative to the JSON file's parent folder.
  - If this field is unspecified, a LUT will be generated from the scanner's properties.
- `axialFOV` : Axial Field-of-View (mm). **[Mandatory]**
- `crystalSize_trans` : Crystal size in the transaxial dimension (mm). **[Mandatory]**
- `crystalSize_z` : Crystal size in the axial dimension (mm). **[Mandatory]**
- `crystalDepth` : Crystal size in the direction of its orientation (mm). **[Mandatory]**
- `scannerRadius` : Scanner radius (mm). **[Mandatory]**.
  - This represents the distance between detector blocks and the isocenter.
- `detsPerRing` : Number of detectors per ring. **[Mandatory]**
- `numRings` : Number of rings. **[Mandatory]**
- `numDOI` : Number of DOI layers. **[Mandatory]**
- `detsPerBlock` : Number of detectors per block in the transaxial dimension.
  - This property is ignored if a LUT is provided

The following properties are necessary to define sensitivity images and histogram shapes:

- `maxRingDiff` : Maximum ring difference two detectors must have to define a valid LOR. **[Mandatory]**
- `minAngDiff` : Minimum difference two detectors must have to define a valid LOR. Has to be an even number. **[Mandatory]**

The following properties are used only in scatter estimation:

- `collimatorRadius` : Collimator radius (mm). Only used in scatter estimation.
- `fwhm` : Energy resolution FWHM (keV). Only used in scatter estimation.
- `energyLLD` : Energy Low-Level-Discriminant (keV). Only used in Scatter estimation.

An optional field can be used to mask detectors:

- `detMask` : Path to mask file. The mask file stores one boolean per detector (in the same order as the LUT file): `true` if the detector is acive, `false` if it is masked. The file is store in binary format, without header, and with one byte per detector.

8.1.1 Example

Here's an example of a Scanner's JSON file

```
{
  "VERSION": 3.2,
  "scannerName": "myscanner",
  "detCoord": "myscanner.lut",
  "axialFOV": 250.0,
  "crystalSize_z": 1.0,
  "crystalSize_trans": 1.0,
  "crystalDepth": 8.0,
  "scannerRadius": 130.0,
  "detsPerRing": 800,
  "numRings": 150,
  "numDOI": 2,
  "maxRingDiff": 50,
  "minAngDiff": 230,
  "detsPerBlock": 40,
  "fwhm": 0.2,
  "energyLLD": 400,
  "collimatorRadius": 100
}
```

8.2 Look-Up-Table

The LUT is a binary file that defines all the *detecting elements*, which can either be individual crystals, or, in the case of scanners with Depth-Of-Interation (DOI), detection positions.

The number of detection elements in the LUT must match the following:

```
detsPerRing * numRings * numDOI
```

For each element, the LUT defines, in 32-bit float:

- X, Y, Z center position of the element
- X, Y, Z orientation (unit vector) of the element, pointing towards the exterior of the scanner

```
X Position of detector 0 (float32)
Y Position of detector 0 (float32)
Z Position of detector 0 (float32)
X Orientation of detector 0 (float32)
Y Orientation of detector 0 (float32)
Z Orientation of detector 0 (float32)
X Position of detector 1 (float32)
Y Position of detector 1 (float32)
Z Position of detector 1 (float32)
X Orientation of detector 1 (float32)
```

(continues on next page)

(continued from previous page)

```
Y Orientation of detector 1 (float32)
Z Orientation of detector 1 (float32)
...
```

The LUT's elements should be ordered in the following way:

- Position in the ring in either clockwise or anti-clockwise
- Ring position, either ascending or descending order
- DOI position, from inner to outer layers.

8.2.1 For Python users

Due to the simplicity of this format, it can be read using the following lines:

```
import numpy as np

lut = np.fromfile("<myscanner>.lut", dtype=np.float32).reshape((-1, 6))
```

Then, one can use matplotlib to display the scanner's detector positions:

```
#<>
import matplotlib.pyplot as plt
N = 800 # Example: The number of crystals per ring is 800
plt.scatter(lut[:N, 0], lut[:N, 1])
```

Or the scanner's detector orientations:

```
#<>
plt.plot(lut[:N, 3])  # X orientation
plt.plot(lut[:N, 4])  # Y orientation
```

Here, `N` is the number of detectors in a ring.

To display the scanner in three dimensions, the repository `plot_scanner` can do that using VTK.

One can also generate the LUT using Python and save it using:

```
lut.tofile("<mynewscanner>.lut")
```

Note: The code above assumes the data type of `lut` to be `np.float32`.

8.2.2 For plugin developers

The value returned by the functions `getDetector1`, `getDetector2`, and `getDetectorPair` should correspond to the index of the detector(s) in that LUT.

CHAPTER

NINE

IMAGE

9.1 Image parameters

The image parameters file is defined in JSON format and looks like the following:

```
{
  "VERSION": 1.0,
  "nx": 192,
  "ny": 192,
  "nz": 89,
  "nt" : 1,
  "vx": 2.0,
  "vy": 2.0,
  "vz": 2.8,
  "off_x": 0.0,
  "off_y": 0.0,
  "off_z": 0.0
}
```

The properties are:

- `nx`, `ny`, `nz` is the image size in X, Y, and Z in number of voxels
- `nt` is the time dimension, which is used for dynamic reconstructions
- `vx`, `vy`, `vz` is the voxel size in X, Y, and Z in millimeters
- `off_x`, `off_y`, `off_z` is the X, Y and Z position of the *center* of the image.
- Optionally, one can also specify:
  - `length_x`, `length_y`, `length_z` to define the physical size of the image in millimeters.
  - If this is not defined, the lengths are computed as `nx*vx`, `ny*vy`, `nz*vz` respectively.

The physical position of any voxel in the X dimension is:

$$x_p = \left(x_i - \frac{n_x - 1}{2} \right) v_x + o_x$$

Where o_x is `off_x` and n_x is `nx`. x_i is the *logical* index of the voxel while x_p is the *physical* position (in millimeters) of the voxel in dimension X. The above equation also applies in dimensions Y and Z.

9.2 Image file format

Images are stored and read in NIfTI format. Since NifTI files include image orientation, origin and pixel spacing, The following requirements are imposed:

- Input images (e.g. attenuation map, sensitivity image) must have an identity orientation matrix (no rotation)
- In the reconstruction script, the origin/pixel size information is read from an image parameter file (described above), unless a sensitivity image is provided (with `--sens`), in which case the orientation/origin/pixel size from the NifTI sensitivity image is used.

9.3 Fourth Dimension (Time)

The image can have a fourth dimension for time, enabling dynamic PET reconstructions. The `nt` field in `ImageParams` sets the number of temporal frames. This is used in conjunction with *Dynamic Framing* to reconstruct an image for each time range.

The image dimensions are `(nt, nz, ny, nx)`. Note that time is the first dimension.

9.3.1 Notes

When using dynamic framing:

- Each frame is reconstructed separately (i.e., independently of other frames) but the iterative updates are calculated for all frames simultaneously
- Sensitivity images can also be 4D when performing a dynamic reconstruction

9.4 For Python users and plugin developers

Note that the `ImageOwned` class has three constructors. One with a single `ImageParams` object, one with a single `string` for the filename, and one with both. The one that only takes a filename deduces the image parameters from the NIfTI header while the one that takes both the NIfTI file and the `ImageParams` object uses the latter to perform a consistency check in order to avoid mismatches.

CHAPTER TEN

LIST-MODE FILE

The default YRT-PET list-mode file is a record of all the events to be considered for the reconstruction.

```
Timestamp for event 0 (ms) (uint32)
Detector1 of the event 0 (uint32)
Detector2 of the event 0 (uint32)
TOF position of event 0 (ps) (float32) [Optional]
Randoms estimate of event 0 (counts/s) (float32) [Optional]
Timestamp for event 1 (ms) (uint32)
Detector1 of the event 1 (uint32)
Detector2 of the event 1 (uint32)
TOF position of event 1 (ps) (float32) [Optional]
Randoms estimate of event 1 (counts/s) (float32) [Optional]
...
```

The file extension used is `.lmDat` by convention. The detectors specified in the List-Mode correspond to the indices in the scanner's LUT.

If the ListMode file contains time-of-flight (TOF) information, the option `--flag_tof` must be used in the executable(s). The TOF value is the difference of arrival time between detector 2 (t_2) and detector 1 (t_1), or $t_2 - t_1$, expressed in picoseconds.

If the ListMode file contains a randoms estimate for each event, the option `--flag_randoms` must be used in the executable(s). The randoms estimate is in counts per second (cps).

10.1 For Python users

If using python bindings, here's how to read a ListModeLUT:

```python
import pyyrtpet as yrt

scanner = yrt.Scanner("<myscanner>.json")
flag_tof = True  # Indicate whether the list-mode file contains a TOF field
flag_randoms = False  # Indicate whether it contains randoms estimates
lm = yrt.ListModeLUTOwned(scanner, "<mylistmode>.lmDat",
                          flag_tof=flag_tof, flag_randoms=flag_randoms)
```

The `flag_tof` option specifies if the list-mode contains TOF information for each event and the `flag_randoms` option specifies if the list-mode contains randoms estimates.

CHAPTER

ELEVEN

LIST-MODE DOI FILE

A more advanced List-Mode format is available for scanners with many DOI layers. The format is similar to the regular List-mode format described above with DOI layer (encoded in 256 bits) for each detector (from the inward face of the detector). The `num_layers` option in the reconstruction executable allows binning of the DOI layers from 256 layers to an arbitrary number of layers.

Note that it is still possible to use the default list-mode format for DOI-enabled scanners. This additional format only allows to save some disk space in case the amount of DOI layers in the scanner is configurable.

```
Timestamp for event 0 (ms) (uint32)
Detector1 of the event 0 (uint32)
DOI1 of the event 0 (uint8)
Detector2 of the event 0 (uint32)
DOI2 of the event 0 (uint8)
TOF position of event 0 (ps) (float32) [Optional]
Randoms estimate of event 0 (counts/s) (float32) [Optional]
Timestamp for event 1 (ms) (uint32)
Detector1 of the event 1 (uint32)
DOI1 of the event 1 (uint8)
Detector2 of the event 1 (uint32)
DOI2 of the event 2 (uint8)
TOF position of event 1 (ps) (float32) [Optional]
Randoms estimate of event 1 (counts/s) (float32) [Optional]
...
```

CHAPTER

TWELVE

MOTION INFORMATION FILE

Motion is recorded in a CSV file. Each line represents a frame. This is the structure of the CSV:

```
t, r00, r01, r02, tx, r10, r11, r12, ty, r20, r21, r22, tz, e
t, r00, r01, r02, tx, r10, r11, r12, ty, r20, r21, r22, tz, e
t, r00, r01, r02, tx, r10, r11, r12, ty, r20, r21, r22, tz, e
...
```

The timestamp `t` is the starting timestamp of the frame. It is an integer. It is in the *same time reference* as the timestamps stored in the List-Mode file. The units are milliseconds.

The motion is defined by a rotation matrix and a translation vector. The rotation matrix is defined as:

```
r00 r01 r02
r10 r11 r12
r20 r21 r22
```

The translation vector is defined by `tx`, `ty`, and `tz`.

The error value `e` is between 0 and 1.

CHAPTER

THIRTEEN

HISTOGRAM3D FORMAT

This page will describe how the *Histogram3D* structure is organised in YRT-PET.

Semantics: The word "Histogram" refers to an array of which the bins are in logical indices. The word "Sinogram", which are not supported yet in YRT-PET, refers to an array of which the bins point to physical coordinates, regardless of how irregular the scanner is.

This means that Histograms point to real detector pairs present in the scanner.

13.1 File format

The histogram file itself is a "RAWD" type. Meaning that it encodes a header describing the array shape and the array itself in 'C'-contiguous ordering.

The file extension commonly used for a Histogram3D is `.his`.

13.1.1 For Python users

It is also possible to read and write them using Python. The file in `yrt-pet/python/pyyrtpet/_pyyrtpet.py` contains a class named `DataFileRawd`, which allows one to read and write in this filetype.

Moreover, with the Python bindings, it is possible to use the fact that the Histogram3D class respects the Python buffer protocol:

```
import pyyrtpet as yrt
import numpy as np

scanner = yrt.Scanner("<myscanner>.json")
his = yrt.Histogram3DOwned(scanner, "<myhistogram>.his")
np_his = np.array(his, copy=False)  # np_his is now a 3D numpy array
```

13.1.2 For MATLAB users

One can work with those files in MATLAB using `read_rawd.m` and `write_rawd.m` in the `scripts/matlab` folder.

13.2 Array format

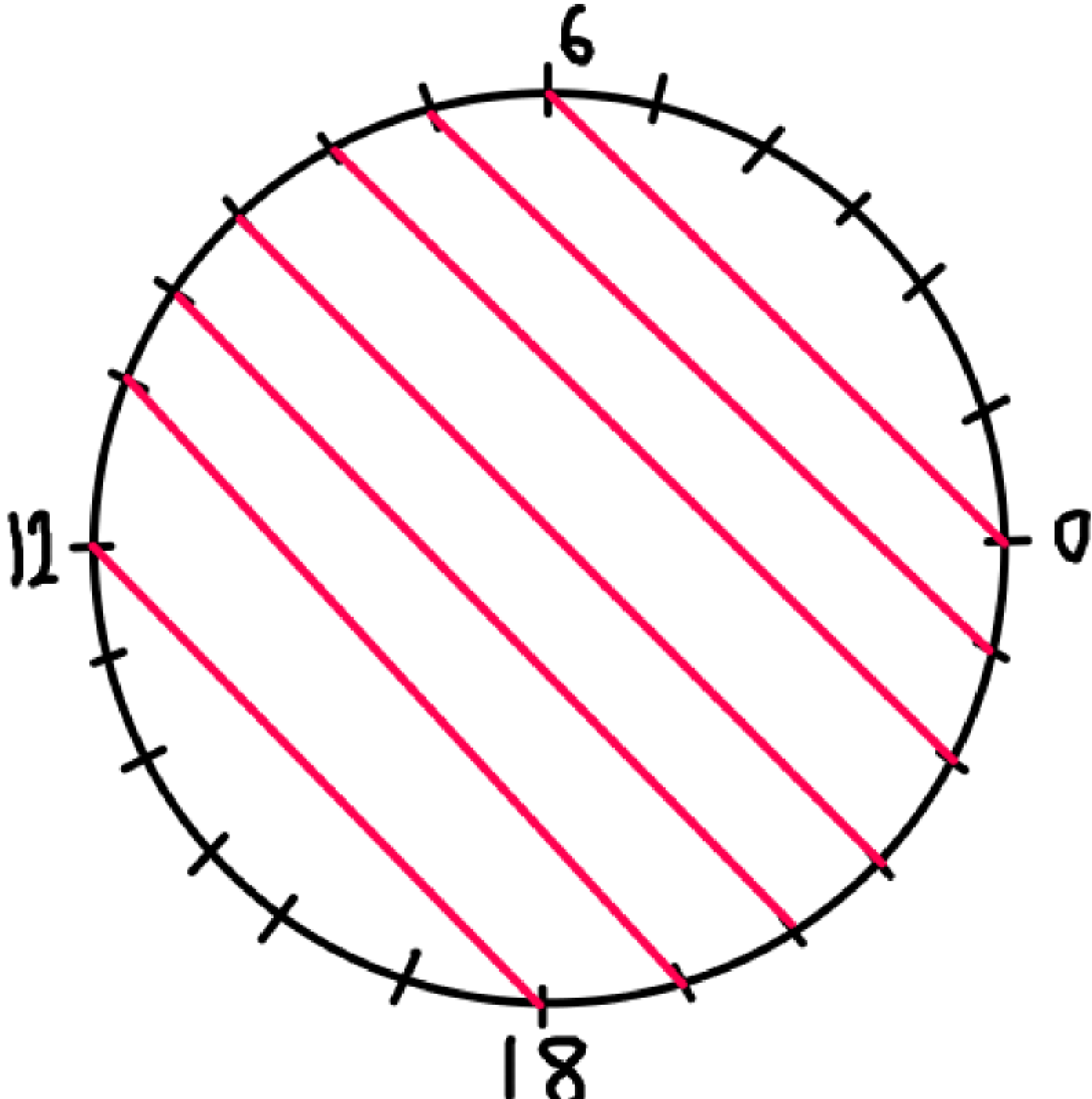
Let's define what a fully 3D histogram is. It is a 3D array such that:

1. Every bin of the histogram, defined by 3 coordinates, stores a particular Line of Response, defined by a pair of detectors
2. Every bin of the histogram must represent a **different** pair of detectors

3. Every Line of Response allowed by the Scanner geometry must be represented by a bin in the histogram
4. Two different lines of responses cannot be represented by the same histogram bin

These rules must apply for a fully 3D scanner with DOI crystals. Note that Time-of-flight bins are not encoded by Histogram3D.

13.3 Crystals in the same ring



Using a certain value of integers `r` and `phi`, we calculate the coordinates of two detectors in the same ring to respect rules 1 and 2:

$$\rho = \begin{cases} 0, & \text{when } \phi \text{ is even} \\ 1, & \text{when } \phi \text{ is odd} \end{cases}$$

$$d_{r1} = r - \frac{n}{4} + \frac{M_a}{2}$$

$$d_{r2} = \frac{n}{2} - (r - \frac{n}{4} + \frac{M_a}{2}) + \rho$$

$$d_1 = (d_{r1} + \frac{\phi}{2}) \mod n$$

$$d_2 = (d_{r2} + \frac{\phi}{2}) \mod n$$

Where:

d_1 and d_2 are detectors 1 and 2 of the bin

n is the number of detectors in the ring

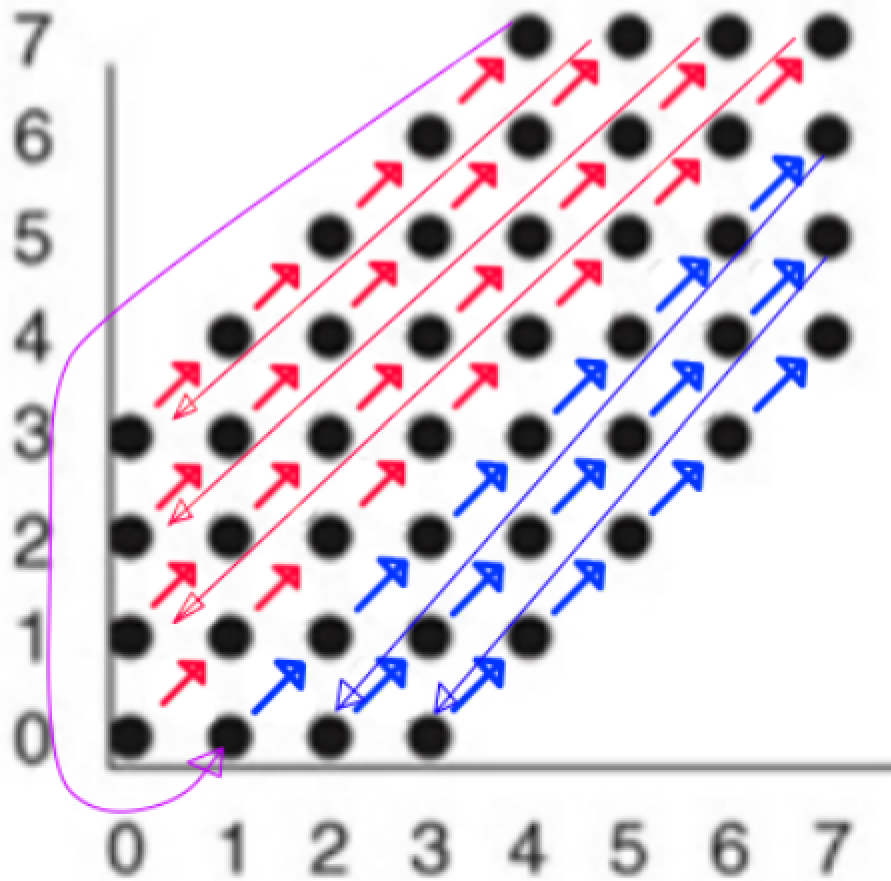
$M_a$is the minimum angular difference of the Scanner Field of view in terms of detector indices

In order to respect rules 3 and 4, a Look-Up-Table of all the pairs `d1` and `d2` and their pair `r` and `phi` is defined for the ring.

This defines a single-ring histogram with no DOI crystals. However, due to the ordering of the detectors in the Look-Up-Table, it is possible to scale this to different axial positions and different DOI layers.

13.4 Binning for different rings

The fully 3D nature of today's scanners makes this task more complex as one LOR can start from a ring and finish in another. To solve this issue, let's define `z_bin`, which represents the position of the LOR in the Michelogram moving diagonally and then from a `delta_z` to another:



$$\Delta z = |z_1 - z_2|$$

$$R_b = \frac{(\Delta z - 1)\Delta z}{2}$$

$$z_{bin} = n_a \Delta z + \min(z_1, z_2) + R_b$$

Where:

z_1 and z_2 are the ring index of detector 1 and 2

n_a is the total number of rings in the scanner

I will spare the inverse function for this document. Note that this equation only accounts for the top left half of the drawing, the other half is managed separately afterward.

13.5 Dealing with multiple DOI layers

A scanner with DOI layers allows for a Line of response to go from a layer to another. To solve this issue, the `r` coordinate of the histogram is used to store this information.

r	Layers for LOR
`r+0`	{0,0}
`r+1`	{0,1}
`r+2`	{1,0}
`r+3`	{1,1}

This has the only disadvantage of making slightly odd plottings when the histogram is shown without taking this into account.

13.6 Histogram Dimensions

The dimensions of the histogram are as such:

$$N_r = N_D^2 * (\frac{N_d}{2} + 1 - M_a)$$

$$N_\phi = N_d$$

$$N_{z_{\text{bin}}} = 2 * \left((M_r + 1) * N_p - \frac{M_r * (M_r + 1)}{2} \right) - N_p$$

Where

N_D is the number of DOI layers

N_r is the number of radial bins

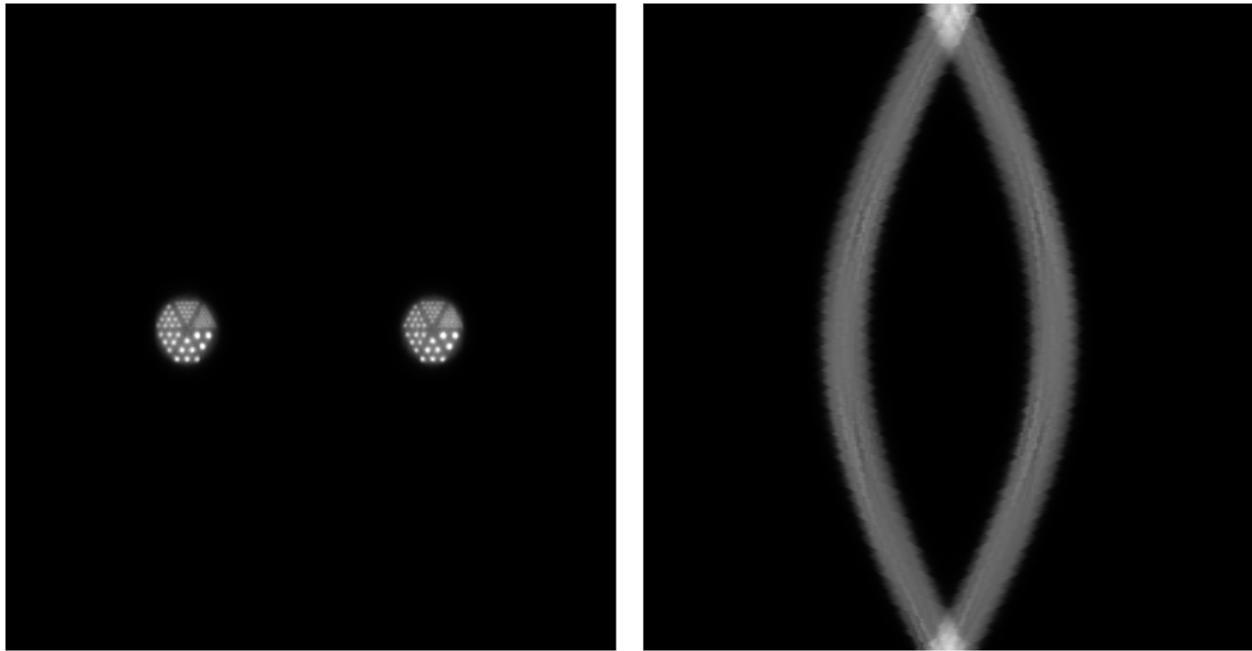
N_d is the number of detectors per ring in the scanner (not counting for DOI)

N_p is the number of axial planes/rings in the scanner

M_a is the minimum angle difference in the ring

M_r is the maximum ring difference in the scanner

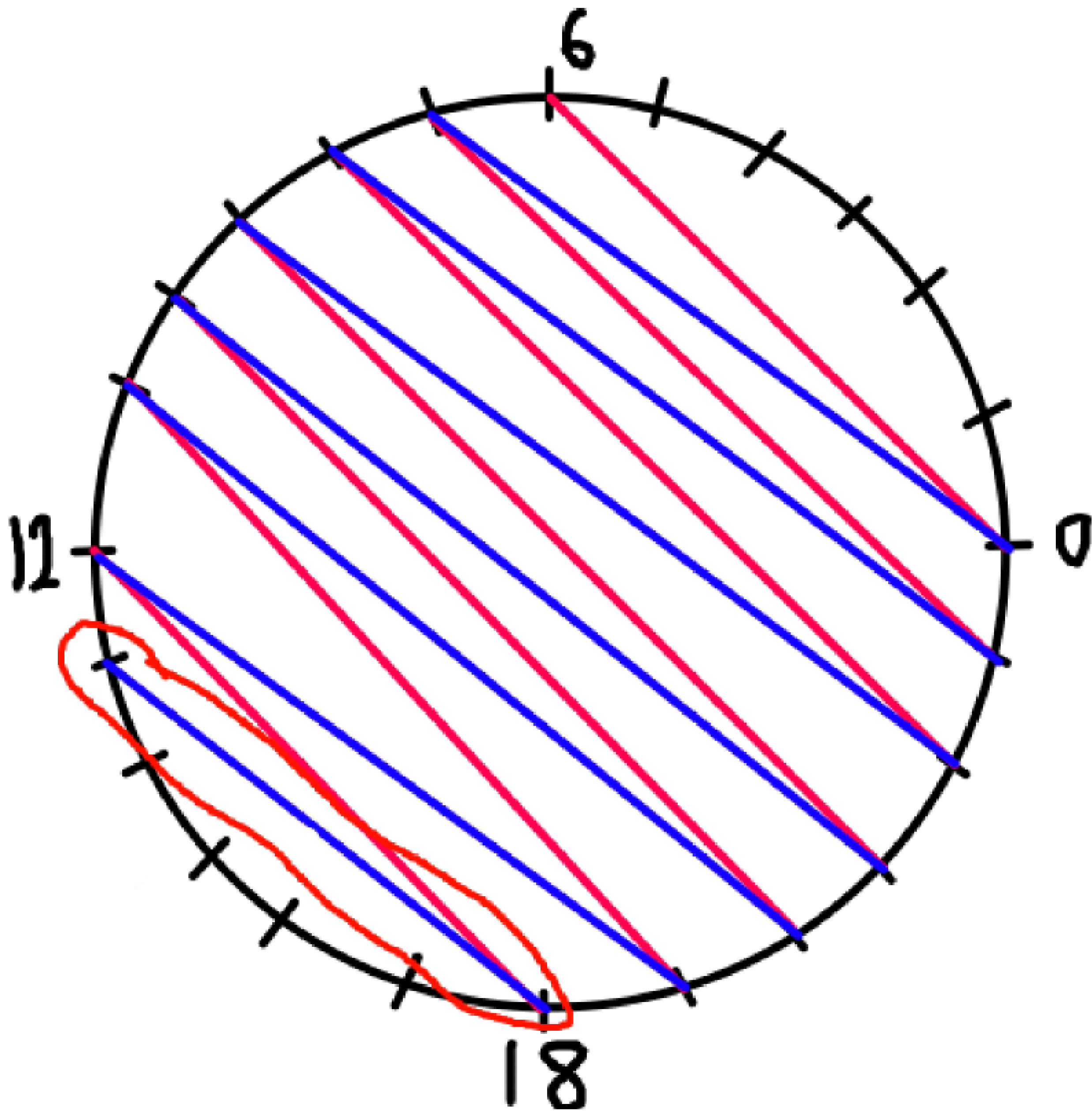
13.7 Example



Left: Image Right: Histogram of the forward projection of the image into a scanner.

13.8 Small asymmetry in the Histogram

Due to the ρ value in the calculation in the same ring, for every odd value of phi, the bin at `r=0` will be an invalid bin for the scanner because it will not respect the "minimum angle difference". It is the circled line in the image below for example:



This is better drawn in "The Theory and Practice of 3D PET" by Bernard Bendriem and David W. Townsend (From Chapter 2 by Michel Defrise and Paul Kinahan):

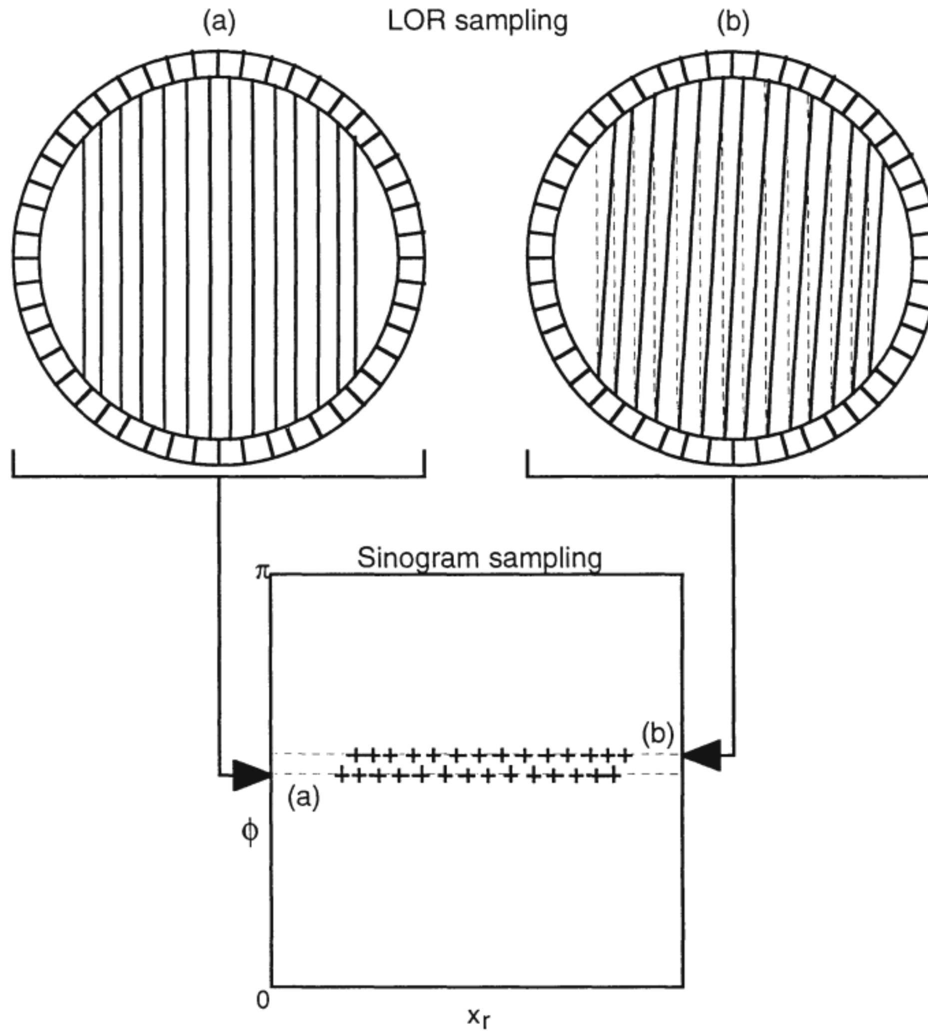


Figure 2.7 sinogram sampling pattern for a scanner with an even number of detector elements. In (b) the LORs are moved by shifting only one of the two detector endpoints, and the previous LORs are indicated by dashed lines. Thus, for (b) an LOR samples the exact center of the FOV, while for (a) the two central LORs straddle the center of the FOV. This leads to an offset in the transverse sampling pattern for adjacent rows.

This does not cause any harm to the reconstruction since these bins will simply never be used. The only harm that it can cause is for the sensitivity image, but these lines are not common at all or are outside the Field of View. If it causes harm to the image, it is still possible to use a Sensitivity histogram as input to the sensitivity image generation and put those bins to zero.

CHAPTER FOURTEEN

RAWD FORMAT

The RAWD file format is defined as follows:

```
MAGIC NUMBER (int32): 732174000 in decimal, used to detect YRT-PET file type
Number of dimensions D (int32)
Dimension 0 (int64)
...
Dimension D - 1 (int64)
Data 0
...
```

Notes:

- The data format is arbitrary and must be known when reading a data file. For instance, images are stored in `float32`.
- The dimensions are ordered with the contiguous dimension last (e.g. Z, Y, X following usual 'C' conventions).
- Just like all binary formats in YRT-PET, the numbers are stored in little-endian.

CHAPTER FIFTEEN

SPARSE HISTOGRAM FILE

Sparse histograms are a list of detector pairs and the value associated with the LOR.

They are structured as follows in the file:

```
Detector1 of the event 0 (uint32)
Detector2 of the event 0 (uint32)
Projection value of the event 0 (float32)
Detector1 of the event 1 (uint32)
Detector2 of the event 1 (uint32)
Projection value of the event 1 (float32)
...
```

The file extension used is `.shis` by convention. The detectors specified in the Sparse histogram correspond to their indices in the scanner's LUT.

15.1 Implementation

Sparse histograms are, in memory, stored as a hash map with the detector pair as a key, ensuring O(1) when accessing projection values.

Currently, the implementation depends on `std::unordered_map`, which tends to be quite slow and is not thread-safe. This will be addressed in the future.

CHAPTER

SIXTEEN

IMAGE-BASED PSF FILE FORMAT

YRT-PET supports image-based point spread function (PSF) input files in two forms:

- **Uniform PSF kernel**
- **Spatially variant PSF kernel (PSF Look-Up Table or LUT)**

Both formats use standard **CSV (comma-separated values)** files.

16.1 Uniform PSF Kernel

16.1.1 Generation

A utility script is provided to generate uniform PSF kernel files:

```
yrt-pet/scripts/utils/generate_psf_kernel.py
```

```
generate_psf_kernel.py [-h] --fx FX --fy FY --fz FZ --vx VX --vy VY --vz VZ
                       [--size_x SIZE_X] [--size_y SIZE_Y] [--size_z SIZE_Z]
                       -o OUTPUT
```

Arguments:

- `--fx FX`: FWHM (full width at half maximum) in X direction (in mm)
- `--fy FY`: FWHM in Y direction
- `--fz FZ`: FWHM in Z direction
- `--vx VX`, `--vy VY`, `--vz VZ`: Voxel sizes in X, Y, and Z (in mm)
- `--size_x SIZE_X`, `--size_y SIZE_Y`, `--size_z SIZE_Z`: (Optional) Kernel sizes in X, Y, and Z (must be odd)
- `-o OUTPUT`, `--output OUTPUT`: Output CSV file path

Output Format:

The resulting CSV file contains:

```
1D convolution kernel in X direction
1D convolution kernel in Y direction
1D convolution kernel in Z direction
Kernel sizes in X, Y, and Z directions
```

This file can be used in YRT-PET reconstruction workflows that support uniform image-based PSF kernels.

16.2 Spatially Variant PSF Kernel (PSF LUT)

YRT-PET also supports **spatially varying 3D symmetric Gaussian kernels**, organized in a structured CSV file (PSF LUT).

16.2.1 Assumptions and Behavior

1. **Symmetry**: PSF kernels are symmetric in X, Y, and Z. Distance to the center is treated as absolute.
2. **Regular Grid**: PSF kernels are placed on a uniform grid. Each "gap" defines the spacing between kernel locations, and the specified "range" must be divisible by the gap.
3. **Interpolation**: Nearest-neighbor interpolation is used to determine which kernel to apply. Out-of-range queries fall back to edge values.
4. **Order**: PSF kernels are stored in the order: X → Y → Z.

16.2.2 PSF LUT CSV Format

```
X,Y,Z range of PSF kernel grid in mm (max offset from center), float
X,Y,Z gap of PSF kernel grid in mm (spacing between kernels), float
X,Y,Z kernel size control (determines how many sigmas are included in kernel), float
SigmaX1,SigmaY1,SigmaZ1  # (kernel at 0,0,0)
SigmaX2,SigmaY2,SigmaZ2  # (kernel at xgap,0,0)
SigmaX3,SigmaY3,SigmaZ3  # (kernel at xgap*2,0,0)
...
```

16.2.3 Example

For:

- XYZ range = 50 mm (maximum distance from the scanner center at which spatially varying PSF kernels are sampled in each direction, forming the outer boundary of the PSF LUT grid.)
- XYZ gap = 50 mm (distance between each kernel center)
- XYZ kernel size control = 4 (number of standard deviation for the gaussian kernels)

Kernels are located in these positions:

```
(0,0,0)
(50,0,0)
(0,50,0)
(50,50,0)
(0,0,50)
(50,0,50)
(0,50,50)
(50,50,50)
```

The file would be structured like this:

```
50,50,50
50,50,50
4,4,4
sigmaX1,sigmaY1,sigmaZ1
sigmaX2,sigmaY2,sigmaZ2
sigmaX3,sigmaY3,sigmaZ3
sigmaX4,sigmaY4,sigmaZ4
sigmaX5,sigmaY5,sigmaZ5
sigmaX6,sigmaY6,sigmaZ6
sigmaX7,sigmaY7,sigmaZ7
sigmaX8,sigmaY8,sigmaZ8
```

CHAPTER

SEVENTEEN

ARRAY

The Array classes provide multidimensional arrays for storing data with support for numpy memory aliasing.

17.1 Overview

Arrays are available from 1D to 5D with two memory management variants:

- **Owned**: Allocates and manages its own memory
- **Alias**: References external memory (e.g., memory owned by numpy arrays)

17.2 Supported Types

- `Array{ND}Float{Owned/Alias}` - Single precision float
- `Array{ND}Double{Owned/Alias}` - Double precision float
- `Array{ND}Int{Owned/Alias}` - Signed integer
- `Array{ND}Bool{Owned/Alias}` - Boolean

Replace `{ND}` with 1, 2, 3, 4, or 5 for the dimension. Replace `{Owned/Alias}` by either `Owned` for memory managed by YRT-PET or `Alias` for memory managed by NumPy.

17.3 Memory management

```
import pyyrtpet as yrt
import numpy as np

# ============================================================================
# Owned array - allocates its own memory
# ============================================================================

# Create a 3D owned array (single precision float)
arr_3d = yrt.Array3DFloatOwned()
arr_3d.allocate([10, 20, 30])

# Verify memory is allocated
assert arr_3d.isMemoryValid(), "Memory should be allocated"

# Get dimensions
dims = arr_3d.getDims()
```

(continues on next page)

(continued from previous page)

```
assert dims[0] == 10 and dims[1] == 20 and dims[2] == 30

# Get total size
total_size = arr_3d.getSizeTotal()
assert total_size == 10*20*30, f"Expected 6000, got {total_size}"

# ============================================================================
# Alias array - binds to numpy array
# ============================================================================

# Create a numpy array
arr_np = np.random.random((5, 10, 20, 30)).astype(np.float32)

# Bind to a NumPy array
arr_alias = yrt.Array4DFloatAlias()
arr_alias.bind(arr_np)

# Change value in numpy array
arr_np[1, 2, 3, 4] = np.float32(-1.2)

# Or Bind from another array
arr_alias2 = yrt.Array4DFloatAlias()
arr_alias2.bind(arr_alias)

# Verify memory is valid
assert arr_alias.isMemoryValid(), "Memory should be valid after binding"
assert arr_alias2.isMemoryValid()
assert arr_alias[1,2,3,4] == arr_np[1,2,3,4], "Memory aliasing failed"
assert arr_alias2[0,2,15,26] == arr_alias[0,2,15,26], "Memory aliasing failed"
```

17.4 Data Access

```
# Create a 3D array for demonstration
arr = yrt.Array3DFloatOwned()
arr.allocate([3, 4, 5])

# Multi-dimensional access (3D)
arr[0, 0, 0] = 1.0
arr[1, 2, 3] = 2.5
value = arr[1, 2, 3]
assert value == 2.5

# Flat access (linear index from 0 to total_size-1)
arr.setFlat(10, 10.0)
value = arr.getFlat(10)
assert value == 10.0

# Increment flat index (useful for parallel operations)
arr.incrementFlat(5, 1.0)  # Adds 1.0 to position 5

# Get dimensions
```

(continues on next page)

(continued from previous page)

```
dims = arr.getDims()
assert len(dims) == 3
assert dims[0] == 3 and dims[1] == 4 and dims[2] == 5

# Get size of a specific dimension
assert arr.getSize(2) == 5 # Size of dim 2, should be 5

# Get strides (step size for each dimension)
strides = arr.getStrides()
# Strides represent how many elements to skip to move to next index in each dim

# Convert between flat and multi-dimensional indices
flat_idx = arr.getFlatIdx([1, 2, 3])
multi_idx = arr.unravelIdx(flat_idx)
assert multi_idx[0] == 1 and multi_idx[1] == 2 and multi_idx[2] == 3
```

17.4.1 Operations

```
# Create and fill array
arr = yrt.Array2DIntOwned()
arr.allocate([10, 10])
arr.fill(0)

# Set some values
arr[5, 5] = 100

# Sum all elements
total_sum = arr.sum()
assert total_sum == 100.0

# Get maximum value
max_val = arr.getMaxValue()
assert max_val == 100.0

# Fill with a new value
arr.fill(5)
assert arr.sum() == 500  # 10*10*5

# Arithmetic operations (in-place)
arr2 = yrt.Array2DIntOwned()
arr2.allocate([10, 10])
arr2.fill(2)

# arr += arr2
arr += arr2
assert arr.getMaxValue() == 7  # 5 + 2

# arr -= scalar
arr -= 1
assert arr.getMaxValue() == 6 # 5 + 2 - 1
```

(continues on next page)

(continued from previous page)

```
# arr *= scalar
arr *= 2
assert arr.getMaxValue() == 12 # (5 + 2 - 1) * 2

# arr /= scalar
arr /= 3
assert arr.getMaxValue() == 4 # ((5 + 2 - 1) * 2) / 3

# Invert array (1/x for each element)
arr = yrt.Array2DDoubleOwned()
arr.allocate([10, 10])
arr.fill(2.0)
arr.invert()
assert abs(arr.getMaxValue() - 0.5) < 0.001

# Copy data from one array to another
arr_src = yrt.Array3DFloatOwned()
arr_src.allocate([5, 5, 5])
arr_src.fill(10.0)

arr_dst = yrt.Array3DFloatOwned()
arr_dst.allocate([5, 5, 5])
arr_dst.copy(arr_src)
assert arr_dst.sum() == 1250.0  # 5*5*5*10
```

17.4.2 File I/O

```
import os

# For this documentation's purposes only, we use a tempfile
import tempfile
with tempfile.NamedTemporaryFile(suffix='.rawd', delete=False) as f:
    my_file = f.name

# Write array to file
arr = yrt.Array3DFloatOwned()
arr.allocate([10, 20, 30])
arr.fill(42.0)
arr.writeToFile(my_file)

# Read array from file
arr2 = yrt.Array3DFloatOwned()
arr2.readFromFile(my_file)

# Verify data
assert arr2.getMaxValue() == 42.0
dims = arr2.getDims()
assert dims[0] == 10 and dims[1] == 20 and dims[2] == 30

# Clean up
os.remove(my_file)
```

17.5 NumPy Interoperability

The Array classes implement the Python buffer protocol, allowing direct numpy conversion:

```
# Create owned array
arr_owned = yrt.Array3DFloatOwned()
arr_owned.allocate([10, 20, 30])

# Create a numpy array with zero-copy
arr_np = np.array(arr_owned, copy=False)

# Modify through numpy
arr_np[:] = 5.0

# arr_owned now contains 5.0
assert arr_owned.getMaxValue() == 5.0

# Alias array works similarly
arr_np_external = np.ones((5, 10, 20), dtype=np.float32)
arr_alias = yrt.Array3DFloatAlias()
arr_alias.bind(arr_np_external)

# You can even bind a numpy array to a YRT-PET array, which is itself bound to
#  a NumPy array.
arr_from_alias = np.array(arr_alias, copy=False)
arr_from_alias[0, 0, 0] = 99.0
assert arr_np_external[0, 0, 0] == 99.0
```

17.6 Boolean Array example

```
# Boolean array
arr_bool = yrt.Array3DBoolOwned()
arr_bool.allocate([5, 5, 5])
arr_bool.fill(False)
arr_bool[0, 0, 0] = True
assert arr_bool[0, 0, 0] == True
```

CHAPTER EIGHTEEN

VECTOR3D AND LINE3D

Vector3D and Line3D are geometric classes for representing 3D vectors and lines in the YRT-PET framework. These classes are used to define Lines of Response (LORs) as 3D lines, which are used in various cases, most notably in projection operations.

18.1 Vector3D

Vector3D represents a 3D vector with single or double precision. By default, we use single precision (32 bits).

18.1.1 Python Usage

```
import pyyrtpet as yrt
import math

# ==========================================================================
# Creating Vectors
# ==========================================================================

# Create a vector with x, y, z components
v = yrt.Vector3D(1.0, 2.0, 3.0)

# Access individual components
x = v.x
y = v.y
z = v.z

assert x == 1.0
assert y == 2.0
assert z == 3.0

# Create a default vector (all zeros)
v_zero = yrt.Vector3D()

# ==========================================================================
# Vector Operations
# ==========================================================================

v1 = yrt.Vector3D(1.0, 2.0, 3.0)
v2 = yrt.Vector3D(4.0, 5.0, 6.0)
```

(continues on next page)

(continued from previous page)

```
# Addition
v_sum = v1 + v2
assert v_sum.x == 5.0
assert v_sum.y == 7.0
assert v_sum.z == 9.0

# Subtraction
v_diff = v2 - v1
assert v_diff.x == 3.0
assert v_diff.y == 3.0
assert v_diff.z == 3.0

# Scalar multiplication
v_mult = v1 * 2.0
assert v_mult.x == 2.0
assert v_mult.y == 4.0
assert v_mult.z == 6.0

# Scalar addition
v_add = v1 + 1.0
assert v_add.x == 2.0
assert v_add.y == 3.0
assert v_add.z == 4.0

# Scalar subtraction
v_sub = v1 - 1.0
assert v_sub.x == 0.0
assert v_sub.y == 1.0
assert v_sub.z == 2.0

# Scalar division
v_div = v1 / 2.0
assert v_div.x == 0.5
assert v_div.y == 1.0
assert v_div.z == 1.5

# ============================================================================
# Vector Properties and Methods
# ============================================================================

v = yrt.Vector3D(3.0, 4.0, 0.0)

# Get the Euclidean norm (length) of the vector
norm = v.getNorm()
assert abs(norm - 5.0) < 0.001, "3-4-0 triangle has norm 5"

# Normalize the vector (make it a unit vector)
v_normalized = v.getNormalized()
assert abs(v_normalized.getNorm() - 1.0) < 0.001

# In-place normalization
v_to_normalize = yrt.Vector3D(3.0, 4.0, 0.0)
```

(continues on next page)

(continued from previous page)

```
v_to_normalize.normalize()
assert abs(v_to_normalize.getNorm() - 1.0) < 0.001

# Check if vector is normalized
assert v_normalized.isNormalized() == True
assert v_to_normalize.isNormalized() == True
assert v.isNormalized() == False

# Update vector components
v = yrt.Vector3D(1.0, 2.0, 3.0)
v.update(10.0, 20.0, 30.0)
assert v.x == 10.0
assert v.y == 20.0
assert v.z == 30.0

# Update from another vector
v1 = yrt.Vector3D(1.0, 2.0, 3.0)
v2 = yrt.Vector3D(4.0, 5.0, 6.0)
v1.update(v2)
assert v1.x == 4.0
assert v1.y == 5.0
assert v1.z == 6.0

# ============================================================================
# Dot product and cross product
# ============================================================================

v1 = yrt.Vector3D(1.0, 0.0, 0.0)
v2 = yrt.Vector3D(0.0, 1.0, 0.0)

# Dot product
dot = v1.x * v2.x + v1.y * v2.y + v1.z * v2.z
assert dot == 0.0  # Orthogonal vectors

# Cross product (vector product)
# Note: We use the multiplication operator (*) for cross product (not dot product)
cross = v1 * v2  # Using the * operator for cross product
assert cross.x == 0.0
assert cross.y == 0.0
assert cross.z == 1.0

# ============================================================================
# Comparison and Equality
# ============================================================================

v1 = yrt.Vector3D(1.0, 2.0, 3.0)
v2 = yrt.Vector3D(1.0, 2.0, 3.0)
v3 = yrt.Vector3D(4.0, 5.0, 6.0)

assert (v1 == v2) == True
assert (v1 == v3) == False
```

(continues on next page)

(continued from previous page)

```
# ===========================================================================
# String Representation
# ===========================================================================

v = yrt.Vector3D(1.5, 2.5, 3.5)
repr_str = str(v)
# Output: (1.5, 2.5, 3.5)
assert "1.5" in repr_str
assert "2.5" in repr_str
assert "3.5" in repr_str

# ===========================================================================
# Double Precision Version
# ===========================================================================

# For higher precision calculations, use Vector3DDouble
v_double = yrt.Vector3DDouble(1.0, 2.0, 3.0)
norm_double = v_double.getNorm()
```

18.2 Line3D

Line3D represents a 3D line segment defined by two endpoints (point1 and point2).

18.2.1 Python Usage

```
import pyyrtpet as yrt
import math

# ===========================================================================
# Creating Lines
# ===========================================================================

# Create a line from two Vector3D points
p1 = yrt.Vector3D(0.0, -100.0, 0.0)
p2 = yrt.Vector3D(0.0, 100.0, 0.0)
line = yrt.Line3D(p1, p2)

# Access the endpoints
assert line.point1.x == 0.0
assert line.point1.y == -100.0
assert line.point2.x == 0.0
assert line.point2.y == 100.0

# Create a default line (both points at origin)
line_default = yrt.Line3D()

# ===========================================================================
# Line Properties
# ===========================================================================

# Get the length (distance between endpoints)
```

(continues on next page)

(continued from previous page)

```
length = line.getNorm()
assert abs(length - 200.0) < 0.001

# ============================================================================
# Line Operations
# ============================================================================

# Update line endpoints
p1_new = yrt.Vector3D(-50.0, -50.0, -50.0)
p2_new = yrt.Vector3D(50.0, 50.0, 50.0)
line.update(p1_new, p2_new)

assert line.point1.x == -50.0
assert line.point2.x == 50.0

# ============================================================================
# Line Comparison
# ============================================================================

# Check if two lines are equal (same endpoints)
line1 = yrt.Line3D(yrt.Vector3D(0, 0, 0), yrt.Vector3D(1, 1, 1))
line2 = yrt.Line3D(yrt.Vector3D(0, 0, 0), yrt.Vector3D(1, 1, 1))
line3 = yrt.Line3D(yrt.Vector3D(1, 1, 1), yrt.Vector3D(2, 2, 2))

assert line1.isEqual(line2) == True
assert line1.isEqual(line3) == False

# ============================================================================
# Line Parallelism
# ============================================================================

# Check if two lines are parallel
line_a = yrt.Line3D(yrt.Vector3D(0, 0, 0), yrt.Vector3D(1, 0, 0))
line_b = yrt.Line3D(yrt.Vector3D(0, 1, 0), yrt.Vector3D(1, 1, 0))
line_c = yrt.Line3D(yrt.Vector3D(0, 0, 0), yrt.Vector3D(0, 0, 1))

assert line_a.isParallel(line_b) == True  # Both along x-axis
assert line_a.isParallel(line_c) == False  # Different directions

# ============================================================================
# Tuple Conversion
# ============================================================================

# Convert line to tuple format
p1 = yrt.Vector3D(1.0, 2.0, 3.0)
p2 = yrt.Vector3D(4.0, 5.0, 6.0)
line = yrt.Line3D(p1, p2)

# toTuple returns ((x1, y1, z1), (x2, y2, z2))
tup = line.toTuple()
assert tup == ((1.0, 2.0, 3.0), (4.0, 5.0, 6.0))
```

(continues on next page)

(continued from previous page)

```
# ============================================================================
# String Representation
# ============================================================================

line = yrt.Line3D(yrt.Vector3D(1.0, 2.0, 3.0), yrt.Vector3D(4.0, 5.0, 6.0))
repr_str = str(line)
# Output contains both points

# ============================================================================
# Double Precision Version
# ============================================================================

# For higher precision, use Line3DDouble
p1d = yrt.Vector3DDouble(0.0, -100.0, 0.0)
p2d = yrt.Vector3DDouble(0.0, 100.0, 0.0)
line_double = yrt.Line3DDouble(p1d, p2d)
```

18.3 Practical Example: Creating LORs for Projection

```
import pyyrtpet as yrt

# Create a scanner with a regular geometry (without specifying a custom a LUT)
scanner = yrt.Scanner(
    scanner_name='EXAMPLE',
    axial_fov=100.0,
    crystal_size_z=2.0,
    crystal_size_trans=2.0,
    crystal_depth=10.0,
    scanner_radius=200.0,
    dets_per_ring=64,
    num_rings=10,
    num_doi=1,
    max_ring_diff=9,
    min_ang_diff=1,
    dets_per_block=16
)

# Gather the 17th detector's position
p1 = scanner.getDetectorPos(17)

# Gather another detector
p2 = scanner.getDetectorPos(142)

# Build an LOR
lor = yrt.Line3D(p1, p2)

# You can then use this LOR with the Siddon or DD projector
# for forward/backward projections
```

18.4 Notes

- Both Vector3D and Line3D support single (float) and double precision versions
- The single precision versions are named `Vector3D` and `Line3D`
- The double precision versions are named `Vector3DDouble` and `Line3DDouble`
- Line3D stores two Vector3D endpoints, so any vector operations can be performed on `line.point1` and `line.point2` individually

CHAPTER

NINETEEN

PROJECTOR

The Projector classes compute a forward or a backward projection for individual LORs (Line of Response). A forward projection projects a line on an image grid and sums the intersecting voxels. A backprojection distributes a given value into the image grid.

19.1 Available Projector Types

YRT-PET provides two projector implementations:

- **ProjectorSiddon** - Siddon projector, faster but less accurate since it does not model the crystal thickness (except with the multi-ray Siddon).
- **ProjectorDD** - Distance-Driven projector, more accurate but slower.

19.2 Python Usage

```
import pyyrtpet as yrt
import numpy as np

# Define a scanner using geometric parameters
scanner = yrt.Scanner(
    scanner_name='MYSCANNER',
    axial_fov=25.0,           # Axial field of view in mm
    crystal_size_z=2.0,       # Crystal size in axial direction (mm)
    crystal_size_trans=2.0, # Crystal size in transaxial direction (mm)
    crystal_depth=10.0,       # Crystal depth (mm)
    scanner_radius=161.0,     # Scanner radius (mm)
    dets_per_ring=256,        # Number of detectors per ring
    num_rings=8,              # Number of detector rings
    num_doi=1,                # Number of DOI layers
    max_ring_diff=7,          # Maximum ring difference
    min_ang_diff=1,           # Minimum angular difference (in number of crystals)
    dets_per_block=32         # Number of crystals per block (transaxial)
)

# Define image grid parameters
img_params = yrt.ImageParams(
    nx=64,     # Number of voxels in x
    ny=64,     # Number of voxels in y
    nz=32,     # Number of voxels in z
    length_x=scanner.scannerRadius*2,
```

(continues on next page)

(continued from previous page)

```
    length_y=scanner.scannerRadius*2,
    length_z=scanner.axialFOV
)

# Create an empty image
image = yrt.ImageOwned(img_params)
image.allocate()
image.fill(0.0)

# Create Projector Parameters
proj_params = yrt.ProjectorParams(scanner)
proj_params.addTOF(300.0, 3)
# Set the projector parameters here

# Create a Distance-Driven projector
projector = yrt.ProjectorDD(proj_params)

# Define a Line of Response (LOR) - a line connecting two detectors
# Define two points representing a line through the FOV
p1 = yrt.Vector3D(0.0, -scanner.scannerRadius, 0.0)   # Point at the top of the ring
p2 = yrt.Vector3D(0.0, scanner.scannerRadius, 0.0)    # Opposite side
lor = yrt.Line3D(p1, p2)

# For DD projector, we need detector orientation (as unit vectors).
# These are computed from the scanner geometry.
# Here we use simplified unit vectors for demonstration.
n1 = yrt.Vector3D(0.0, -1.0, 0.0)
n2 = yrt.Vector3D(0.0, 1.0, 0.0)

# Perform single backprojection
# Parameters: image, LOR, detector_orient_1, detector_orient_2, projection_value,
#   dynamic_frame, tof_helper, tof_value
# `projection_value` is the value that will be backprojected
# `tof_helper` is an object used to compute the weight of each pixel w.r.t. the TOF kernel
# `tof_value` is the TOF measure in picoseconds to use for the backprojection.
#   We use -50ps here for example
tof_helper = projector.getTOFHelper()
projector.backProjection(
    image, lor, n1, n2, 6, 0, tof_helper, -50
)

# Ensure that we populated at least some pixels
image_np = np.array(image, copy=False)
assert image_np.max() > 0

# Forward projection

#  This time, we will use the Siddon projector
projector = yrt.ProjectorSiddon(proj_params)

# Define a LOR
p1 = yrt.Vector3D(-scanner.scannerRadius, 0.0, 0.0)   # Point at the left of the ring
```

(continues on next page)

(continued from previous page)

```
p2 = yrt.Vector3D(scanner.scannerRadius, 0.0, 0.0)    # Opposite side
lor = yrt.Line3D(p1, p2)

# For the single-ray Siddon projector, we do not need to specify the
#  detector orientation

# Perform single forward projection
tof_helper = projector.getTOFHelper()
forward_proj = projector.forwardProjection(
    image, lor, n1, n2, 0, 0, tof_helper, 0
)

# Ensure we intersected at least some voxels
assert forward_proj > 0
```

19.3 Siddon vs Distance-Driven

The two projector implementations have different characteristics:

Feature	Siddon	Distance-Driven
Speed	Faster	Slower
Accuracy	Lower	Higher
Time-of-Flight support	Yes	Yes
Projection-space PSF	No	Yes
Modeling crystal thickness	Only with multi-ray Siddon	Yes

19.4 Siddon-specific Methods

```
# Siddon projector supports multi-ray configuration
projector_siddon = yrt.ProjectorSiddon(proj_params)

# Set number of rays for multi-ray sampling
projector_siddon.setNumRays(5)  # Use 5 rays per LOR
assert projector_siddon.getNumRays() == 5
```

19.5 Single LOR Projection Methods (Siddon)

The Siddon projector provides static methods for single LOR projections:

```
# Forward projection - returns contribution of image along the LOR
value = yrt.ProjectorSiddon.singleForwardProjection(
    image,      # Input image
    lor,        # Line of Response
    0,          # Dynamic frame index
    None,       # TOF helper (or None)
    0.0         # TOF value
)
```

(continues on next page)

(continued from previous page)

```
# Back projection - adds contribution to image along the LOR
yrt.ProjectorSiddon.singleBackProjection(
    image,      # Output image (modified in-place)
    lor,        # Line of Response
    1.0 ,       # Value to backproject
    0,          # Dynamic frame index
    None,       # TOF helper (or None)
    0.0         # TOF value
)
```

19.6 Notes

- The Siddon projector does NOT support projection-space PSF. If you try to add a PSF to a Siddon projector, it will be ignored with a warning. Use the Distance-Driven projector for PSF-aware projections.
- For a full forward/backward projection on an entire dataset, use `OperatorProjector` See *operator-projector.md* for details

CHAPTER

TWENTY

BIN ITERATOR

Bin iterators define how to iterate over projection data structures.

20.1 Python Types

```
import pyyrtpet as yrt

scanner = yrt.Scanner("<MyScanner.json>")

# Create from histogram
his = yrt.Histogram3DOwned(scanner)
his.allocate()

# Get bin iterator for OSEM subsets (take the fourth out of eight subsets)
bin_iter = his.getBinIter(num_subsets=8, subset_idx=3)

# The bin iterator can then be used with `OperatorProjector` to project a
#  specific subset.
```

20.2 Available Iterators

Here are some of the bin iterator types used in this project

- `BinIteratorRange` - Simple range iteration
- `BinIteratorVector` - Custom list of bins
- `BinIteratorChronological` - Time-ordered iteration
- `BinIteratorChronologicalInterleaved` - Interleaved time-ordered (For list-mode reconstruction)

The most common usage is through `getBinIter()` on histogram or list-mode data. This function will create a bin iterator using the provided arguments and return it.

20.3 For plugin developers

When defining a new histogram data format, one must define the `getBinIter(...)` function in order to allow the new histogram format to be used for reconstruction or projection purposes.

CHAPTER

TWENTYONE

OPERATORPROJECTOR

OperatorProjector wraps a Projector and performs either forward projections or backprojections on a set of LORs provided by a given projection-space dataset (either list-mode or histogram).

21.1 Python Usage

```
import pyyrtpet as yrt
import numpy as np

# ============================================================================
# Step 1: Create a Scanner
# ============================================================================

# Define a scanner using geometric parameters
scanner = yrt.Scanner(
    scanner_name='SOME_SCANNER',
    axial_fov=20.0,             # Axial field of view (mm)
    crystal_size_z=2.0,         # Crystal size in z (mm)
    crystal_size_trans=2.0,     # Crystal size in transaxial (mm)
    crystal_depth=10.0,         # Crystal depth (mm)
    scanner_radius=200.0,       # Scanner radius (mm)
    dets_per_ring=256,          # Detectors per ring
    num_rings=10,               # Number of rings
    num_doi=1,                  # DOI layers
    max_ring_diff=9,            # Max ring difference
    min_ang_diff=1,             # Min angular difference
    dets_per_block=32           # Detectors per block
)

# ============================================================================
# Step 2: Create Image Parameters
# ============================================================================

img_params = yrt.ImageParams.fromParams(
    nx=64,    # Number of voxels in x
    ny=64,    # Number of voxels in y
    nz=11,    # Number of voxels in z
    vx=2.0,   # Voxel size x (mm)
    vy=2.0,   # Voxel size y (mm)
    vz=2.0    # Voxel size z (mm)
```

(continues on next page)

(continued from previous page)

```
)

# ============================================================================
# Step 3: Create Image with random values
# ============================================================================

image = yrt.ImageOwned(img_params)
image.allocate()

# Fill with random values
image_np = np.array(image, copy=False)
image_np_init_id = id(image_np)
image_np[:] = np.random.rand(*image_np.shape).astype(np.float32)

assert id(image_np) == image_np_init_id, "Image should not have been moved"
assert image_np.max() > 0, "Image should have data"

# ============================================================================
# Step 4: Create Projection Data (Histogram3D)
# ============================================================================

# Forward projection: image -> projection data
# The histogram will be modified with the forward projected values
his_fwd = yrt.Histogram3DOwned(scanner)
his_fwd.allocate()
his_fwd.clearProjections(0.0)

# Verify histogram is properly allocated
num_bins = his_fwd.count()
assert num_bins > 0

# ============================================================================
# Step 5: Create Projector Parameters
# ============================================================================

proj_params = yrt.ProjectorParams(scanner)
proj_params.setProjector("DD")  # Use Distance-Driven projector
# Other projector options would go here

# ============================================================================
# Step 6: Create Bin Iterator
# ============================================================================

# Get a bin iterator for the histogram
# Parameters: number of subsets, subset index
bin_iter = his_fwd.getBinIter(num_subsets=1, idx_subset=0)

# ============================================================================
# Step 7: Create OperatorProjector
# ============================================================================

# Create the operator projector with projector params and bin iterator
```

(continues on next page)

(continued from previous page)

```
oper = yrt.OperatorProjector(proj_params, bin_iter)

# ===========================================================================
# Step 9: Forward Projection (Image -> Histogram3D)
# ===========================================================================

# Apply forward projection
oper.applyA(image, his_fwd)

# Check the forward projection results
fwd_np = np.array(his_fwd, copy=False)
fwd_sum = fwd_np.sum()
fwd_max = fwd_np.max()

print(f"Forward projection sum: {fwd_sum}")
print(f"Forward projection max: {fwd_max}")

assert fwd_sum > 0, "Forward projection should produce non-zero values"
assert fwd_max > 0, "Forward projection should have positive values"

# ===========================================================================
# Step 10: Back Projection (Histogram3D -> Image)
# ===========================================================================

# Create empty image for back projection
bp_image = yrt.ImageOwned(img_params)
bp_image.allocate()
bp_image.fill(0.0)

# Create a histogram and populate it with random values
his_for_bp = yrt.Histogram3DOwned(scanner)
his_for_bp.allocate()
his_for_bp_np = np.array(his_for_bp, copy=False)
his_for_bp_np[:] = np.random.rand(*his_for_bp_np.shape).astype(np.float32)

# Apply back projection
oper.applyAH(his_for_bp, bp_image)

# Check the back projection results
bp_image_np = np.array(bp_image, copy=False)
bp_sum = bp_image_np.sum()
bp_max = bp_image_np.max()

print(f"Back projection sum: {bp_sum}")
print(f"Back projection max: {bp_max}")

assert bp_sum > 0, "Back projection should produce non-zero values"
assert bp_max > 0, "Back projection should have positive values"

# Advanced: This is the list of properties gathered by the projection operator for
#  every event or bin
prop_types = oper.getProjectionPropertyTypes()
```

(continues on next page)

(continued from previous page)

```
print(f"Required properties: {prop_types}")
```

21.2 Important methods

- `applyA(image, projection_data)` - Forward projection: image -> projection data
- `applyAH(projection_data, image)` - Back projection: projection data -> image
- `addTOF(tof_width_ps, tof_num_std)` - Add time-of-flight configuration
- `addProjPSF(fname)` - Add projection-space PSF

Note: Configuration methods must be called AFTER creating the OperatorProjector but BEFORE calling any projection operations.

CHAPTER

TWENTYTWO

PROJECTIONLIST

ProjectionList stores list-mode projection values. It references a source (histogram or list-mode) for LOR geometry and stores only the measurement values.

22.1 Python Usage

```python
import pyyrtpet as yrt
import numpy as np

# Define a scanner using geometric parameters
scanner = yrt.Scanner(
    scanner_name='MYSCANNER',
    axial_fov=25.0,          # Axial field of view in mm
    crystal_size_z=2.0,      # Crystal size in axial direction (mm)
    crystal_size_trans=2.0,  # Crystal size in transaxial direction (mm)
    crystal_depth=10.0,      # Crystal depth (mm)
    scanner_radius=161.0,    # Scanner radius (mm)
    dets_per_ring=256,       # Number of detectors per ring
    num_rings=8,             # Number of detector rings
    num_doi=1,               # Number of DOI layers
    max_ring_diff=7,         # Maximum ring difference
    min_ang_diff=1,          # Minimum angular difference (in number of crystals)
    dets_per_block=32        # Number of crystals per block (transaxial)
)

# Create a histogram reference
his = yrt.Histogram3DOwned(scanner)
his.allocate()

# Create the ProjectionList object
proj_list = yrt.ProjectionListOwned(his)
proj_list.allocate()

# Access as numpy array
proj_np = np.array(proj_list, copy=False)  # Shares memory

# Set values
proj_np[:] = 1.0

# Or use Alias to bind to external numpy
```

(continues on next page)

(continued from previous page)

```
proj_np_ext = np.zeros(his.count(), dtype=np.float32)
proj_alias = yrt.ProjectionListAlias(his)
proj_alias.bind(proj_np_ext)

# The ProjectionList object can then be used just like any projection-space data
# object, including with projection operators (as input to applyA and applyAH)

# This is allows for custom algorithm implementations
```

22.2 Owned vs Alias

- `ProjectionListOwned` - Allocates and manages its own memory
- `ProjectionListAlias` - References external array

All member methods (getProjectionValue, setProjectionValue, getDetector1, etc.) delegate to the reference projection data source.

CHAPTER

TWENTYTHREE

OWNED VS ALIAS (FOR PYTHON USERS AND PLUGIN DEVELOPERS)

Almost every data structure class in YRT-PET comes in two forms, one has the `Owned` suffix and the other has the `Alias` suffix.

An `Owned` class is when the inner memory is managed by the object itself. This means that the lifetime of the memory is the lifetime of the object itself. After the creation of an `Owned` object, one has to call the `allocate()` function before reading/writing in its memory. If this is not done, the execution will fail into a segmentation fault.

An `Alias` class is when the memory is not managed by the object, but by an external instance. The object only manages the internal metadata and will not deallocate the memory on destruction. This can be seen as using a reference to another object. The main use of `Alias` objects is when working with numpy arrays in Python, as the example below will demonstrate. Similarly, after the creation of an `Alias` object, one has to call the `bind()` function before reading/writing in its memory. If this is not done, the execution will fail into a segmentation fault.

23.1 Examples

23.1.1 Example usage of `Owned` object

```
import pyyrtpet as yrt
import numpy as np

img_shape = (100, 100, 50)  # in x, y, z dimensions
img_size_mm = (30, 30, 25)  # in x, y, z dimensions
img_params = yrt.ImageParams(*img_shape, *img_size_mm)

# Allocate memory in YRT-PET
img_yrt = yrt.ImageOwned(img_params)
img_yrt.allocate()

# Bind to numpy
img_np = np.array(img_yrt, copy=False)
# Setting "copy=False" is what allows for the memory aliasing.

# From now on, whatever that is done in the img_np array,
# it will internally be done in img_yrt's memory
# and vice-versa

# Example: This will write in img_yrt's memory
img_np[:] = 1 # The "[:]" is important to avoid reassignment
```

(continues on next page)

(continued from previous page)

```
# This will write the image into a file
img_yrt.writeToFile("<my_image.nii>")
```

23.1.2 Example usage of `Alias` object

Since most data structure objects respect the Python buffer protocol, one can bind a numpy array into an `Alias` object.

```
import pyyrtpet as yrt
import numpy as np

img_shape = (100, 100, 50)  # in x, y, z dimensions
img_size_mm = (30, 30, 25)  # in x, y, z dimensions
img_params = yrt.ImageParams(*img_shape, *img_size_mm)

# Allocate memory in numpy
img_np = np.ones([img_params.nz, img_params.ny, img_params.nx], dtype=np.float32)

# Bind to YRT-PET
img_yrt = yrt.ImageAlias(img_params)
img_yrt.bind(img_np)

# Whatever is done in the img_np array,
# it will internally be done in img_np's memory,
# and vice-versa

# Example: This will write in img_yrt's memory
img_np[:, 0, :] = 2

# This will write the image into a file
img_yrt.writeToFile("<my_image.nii>")
```

23.1.3 Example usage of `Alias` object *in GPU*

Just as numpy allows the use of `Alias` objects by managing the CPU memory of a YRT-PET object, PyTorch can do the same for GPU memory, as the example below demonstrates.

```
import torch
import pyyrtpet as yrt

# %% Use CUDA device 0
cuda0 = torch.device('cuda:0')

# %% Ask YRT-PET for the available VRAM
mem_available = yrt.getAvailableVRAM()

# %% Define image parameters
img_shape = (100, 100, 50)  # in x, y, z dimensions
img_size_mm = (100.0, 100.0, 50.0)  # in x, y, z dimensions
params = yrt.ImageParams(*img_shape, *img_size_mm)

# %% Create Torch array and bind it to an ImageDeviceAlias
ones_img = torch.zeros(
```

(continues on next page)

(continued from previous page)

```
    [params.nz, params.ny, params.nx],
    device=cuda0,
    dtype=torch.float32,
    layout=torch.strided,
)
img_dev = yrt.ImageDeviceAlias(params)
# Bind Torch array to YRT-PET Image
img_dev.setDevicePointer(ones_img.data_ptr())

# Now, the "ones_img" Torch array points to the same memory location
# as the "img_dev" YRT-PET ImageDevice object

# %% Initialize the scanner
#  (Here we define a regular scanner using geometric properties, but we could
#  also do it with a JSON file)
scanner = yrt.Scanner(scanner_name='MYSCANNER', axial_fov=25,
                      crystal_size_z=2.0, crystal_size_trans=2.0,
                      crystal_depth=10.0, scanner_radius=161,
                      dets_per_ring=256, num_rings=8, num_doi=1,
                      max_ring_diff=7, min_ang_diff=1, dets_per_block=32)

# %% Initialize an empty histogram
his = yrt.Histogram3DOwned(scanner)
his.allocate()
his.clearProjections(1.0)

# %% Initialize the projector

# Create a bin iterator that uses one subset and iterates on subset 0
bin_iter = his.getBinIter(1, 0)

# Define the projector parameters
proj_params = yrt.ProjectorParams(scanner)
proj_params.setProjector("DD")

# Create the projector
oper = yrt.OperatorProjectorDD_GPU(proj_params)
props = oper.getProjectionPropertyTypes(his)

# %% Create a projection-space device buffer
# Use 'his' as a reference to compute LORs and use 1 OSEM subset
his_dev = yrt.ProjectionListDeviceAlias(scanner, his, props, mem_available, 1)

# Important: This is needed to precompute all LORs and load them into the device
# Arguments: Load events from the batch 0 of the subset 0
his_dev.prepareBatchLORs(0, 0)

# Create a Torch array with the appropriate size
ones_proj = torch.ones(
    [his_dev.getLoadedBatchSize()],
    device=cuda0,
    dtype=torch.float32,
```

(continues on next page)

(continued from previous page)

```
    layout=torch.strided,
)

# Bind Torch array to YRT-PET ProjectionData
his_dev.setProjValuesDevicePointer(ones_proj.data_ptr())

# Do the backprojection on the image
oper.applyAH(his_dev, img_dev)

# Save image
img_dev.writeToFile("<my_image.nii>") # save img
```

CHAPTER

TWENTYFOUR

OSEM

OSEM (Ordered Subsets Expectation Maximization) is the main reconstruction algorithm.

24.1 Basic Usage

This is an example usage performing the OSEM reconstruction of a list-mode dataset.

```
import pyyrtpet as yrt

scanner = yrt.Scanner("<MyScannerFile.json>")

lm = yrt.ListModeLUTOwned(scanner, "<MyListMode.lmDat>")

# Create OSEM with scanner
osem = yrt.OSEM(scanner)

# Set reconstruction parameters
osem.num_MLEM_iterations = 10
osem.num_OSEM_subsets = 8

# Set image parameters
img_params = yrt.ImageParams(nx=128, ny=128, nz=64,
    length_x=128.0, length_y=128.0, length_z=64.0)
osem.setImageParams(img_params)

# Set data input
osem.setDataInput(lm)

# Set projector
osem.setProjector("Siddon")

# Generate sensitivity images (Will also save the sensitivity image into a NIfTI file)
[sens_image] = osem.generateSensitivityImages("sens_image.nii.gz")

# Set sensitivity image(s)
osem.setSensitivityImage(sens_image)

# Run reconstruction (Will also save the reconstructed image into a NIfTI file)
result = osem.reconstruct("recon_image.nii.gz")
```

24.2 Configuration

24.2.1 Reconstruction Parameters

```
#<>
osem.num_MLEM_iterations = 20  # Number of iterations
osem.num_OSEM_subsets = 8      # Number of subsets
```

24.2.2 Data Input

```
#<>
osem.setDataInput(projection_data)
```

24.2.3 Projector

```
#<>
osem.setProjector("DD")      # For the distance-driven projector
osem.setProjector("SIDDON")  # For the Siddon projector

osem.setNumRays(3)                 # Multi-ray (Siddon only)
osem.addProjPSF("<proj_psf.csv>")  # Adding projection-space PSF (DD only)

# Time-of-flight (TOF width in picoseconds, number of standard deviations to fill
#    from the TOF kernel)
osem.addTOF(500.0, 3)
```

24.2.4 Corrections

Sensitivity correction (sometimes also called normalization correction) can be done by providing a histogram (which can be of any format that inherits from `Histogram`, not necessarily `Histogram3D`) of the sensitivity coefficient for every detector pair.

Attenuation correction can be done by providing an attenuation image or by providing a histogram of Attenuation Correction Factors (ACFs).

Randoms correction can be performed by either providing a histogram of the randoms estimate for every detector pair, or by encoding the randoms estimate for every event (overloaded in the `getRandomsEstimate` member function)

Scatter correction (Not to be confused with scatter *estimation*) is perfomed by providing a histogram of the scatter estimate for every detector pair.

```
#<>
osem.setSensitivityHistogram(sensitivity_his)
osem.setAttenuationImage(att_image)    # Attenuation correction (From mu-map)
osem.setACFHistogram(acf_his)          # Attenuation correction (From histogram)
osem.setRandomsHistogram(randoms_his) # Randoms correction (From histogram)
osem.setScatterHistogram(scatter_his) # Scatter correction
```

24.2.5 Point Spread Function (Image-space)

```
#<>
osem.addImagePSF("<psf.csv>")                                # Uniform PSF
osem.addImagePSF("<psf_variant.csv>", yrt.ImagePSFMode.VARIANT)  # Spatially variant PSF
```

PSF Modes

- `yrt.ImagePSFMode.UNIFORM` - Same PSF for all voxels
- `yrt.ImagePSFMode.VARIANT` - Spatially variant PSF

24.2.6 Output Options

To save intermediate iteration, you must provide a `RangeList` object

```
# Save intermediate iterations
range_list = yrt.RangeList()
range_list.insertSorted(0,1) # Save iterations 0 and 1
range_list.insertSorted(5,8) # Save iterations 5,6,7,8
range_list.insertSorted(10,10) # Save iteration 10
```

to the `OSEM` object

```
#<>
# The filename given will be added a prefix to specify which iteration is was saved
osem.setSaveIterRanges(range_list, "./iterations/recon_image_intermediate.nii.gz")

# Custom initial estimate (instead of a uniform image containing the value 0.1)
osem.setInitialEstimate(initial_image)

# Reconstruction mask (To disable some voxels from the reconstruction)
osem.setMaskImage(mask_image)
```

24.3 Output

The `reconstruct()` method returns an `ImageOwned` object, but can also save the reconstructed image to disk if provided a filename.

```
#<>
result = osem.reconstruct() # Will not save to disk
result = osem.reconstruct("recon_image.nii.gz") # Will save to disk

# Access as numpy array
result_np = np.array(result, copy=False)
```

CHAPTER

TWENTYFIVE

CONSTRAINTS

Constraints filter which LORs are included in reconstruction.

25.1 Python Types

```python
import pyyrtpet as yrt

# Minimum angle difference constraint
constraint = yrt.ConstraintAngleDiffDeg(30.0)

# Minimum block difference constraint
constraint = yrt.ConstraintBlockDiffIndex(2)

# Detector mask for a hypothetical scanner with 1000 crystals
detmask = yrt.DetectorMask(1000)
# Detector mask constraint
constraint = yrt.ConstraintDetectorMask(detmask)
```

Constraints are passed to the projector or OSEM to filter LORs during reconstruction.

CHAPTER TWENTYSIX

FREQUENTY ASKED QUESTIONS

Relating to image manipulation:

- I have a NIfTI image and I want to generate the corresponding image parameters JSON file
  - With Python, this one-liner: `yrt.ImageOwned("<my_image.nii>").getParams().serialize("<my_params.json>")`
    - Without Python, this can be done manually quite easily. The only caveat is that the image parameters file needs to specify an *offset*, which is the position of the center of the image. This is different from the *origin* of the image, which is the physical position of the first voxel.
      - $p_{0_x} = o_x - \frac{l_x}{2} + \frac{v_x}{2}$ where p_{0_x} is the origin, o_x is the image offset, l_x is the image length and v_x is the voxel size in the X dimension. The same equation applies in the Y and Z dimensions

Relating to CASToR:

- I have scanner parameters from CASToR and want to use them in YRT-PET
  - For the Look-Up-Table:
    - CASToR provides a tool (`castor-scannerLUTExplorer`) that allows one to display each detecting element
    - The following command line will generate a log file containing a *text* description of each detector: `castor-scannerLUTExplorer -sf <my geom file> -g -o <output log file>`
    - Then, the script in `scripts/data_conversion/convert_CASToR_to_YRT-PET_LUT.py` takes that log file and converts it into a YRT-PET-ready Look-Up-Table.
  - For the Scanner parameters JSON file:
    - Most of the scanner parameters can be determined manually. Though some caveats are worth mentioning:
      - Note that the minimum angle difference (named `minAngDiff` in YRT-PET's JSON file and `min angle difference` in CASToR's hscan file) is described in YRT-PET in terms of *number of detector elements* while in CASToR it is described in degrees.
    - Similarly, the maximum ring difference `maxRingDiff` property is defined in terms of the number of rings.
- I have a List-mode file in the CASToR format, how do I convert it in the YRT-PET default ListMode format (`LM`) ?
  - When there is no Time-of-Flight (TOF) and no Attenuation correction, the CASToR list-mode datatype is equivalent to YRT-PET's.
  - When there is TOF in the List-mode, CASToR's structure encodes the TOF slightly differently. The script in `scripts/convert_CASToR_to_YRT-PET_list-mode.py` does the conversion.

    * Note that the script does not handle normalisation information (CASToR 3.2).
    * YRT-PET's default List-Mode format does not encode as many fields as CASToR's, so a one-to-one equivalence is not to be expected.

- My images seem to be flipped in the Z direction compared to CASToR…
  - This might be because CASToR uses a left-handed coordinate system while YRT-PET uses a right-handed coordinate system.

CHAPTER

TWENTYSEVEN

CONTRIBUTING

27.1 Git feature development workflow

- The main branch is the stable branch of the project, it is designed to remain "clean" (i.e. tested and documented).
- Work on the project should be performed on *branches*. A branch is created for each feature, bug, etc. and progress is committed on that branch.
- Once the work is ready to *merge* (i.e. the code is complete, tested and documented) it can be submitted for review: the process is called *merge request*. The version should be bumped by increasing the version number (major, minor or patch, depending on the changes) in `VERSION.txt`. Follow Github's instructions to submit a pull request.
- The request is then reviewed by one of the project's maintainers. After discussions, it should eventually be merged to the main branch by the maintainer.
- The branch on which the work was performed can then be deleted (the history is preserved). The branch commits will be squashed before the merge (this can be done in Github's web interface).
- If the main branch has changed between the beginning of the work and the pull request submission, the branch should be *rebased* to the main branch (in practice, tests should be run after a rebase to avoid potential regressions).
  - In some cases, unexpected rebase are reported (for instance if the history is A–B–C and B is merged to A, a later rebase of C to A may cause conflicts that should not exist). In such cases, two fixes are possible:
    - Launching an interactive rebase (`git rebase -i <main branch>`) and dropping the commits that would be duplicated.
  - After a rebase, `git push` by default will not allow an update that is not `fast-forward` with the corresponding remote branch, causing an error when trying to push. `git push --force-with-lease` can be used to force a push while checking that the remote branch has not changed. Note that this will lose history

27.2 Testing

There are two types of tests available, unit tests and integration tests.

- Unit tests are short validation programs that check the operation of individual modules. To run the unit test suite, simply run `make test` after compiling (with `make`).
- Integration test data is currently not publicly available.

27.3 Code standard

Aggregation of the basic ideas on how the reconstruction code should be made. Note: Italic was used to highlight part that were still unsure or not decided.

We use clang-format for the code formatting. The variable naming standard goes with the following rules:

- All variables should have the following format: `prefixes_variableNameInCamelCase_suffixes`
- Private or protected member variables should have the prefix `m`
- Pointer variables, both raw and smart, should have the prefix `p`
- Reference variables should have the prefix `r`
- A device object or pointer should have the prefix `d`
- The prefix `h` can be used to highlight that a variable is host-side
- Parameters should have the prefix `p` when the variable can be confused with a local or a member variable
- Suffixes should be used only when adding units to a variable
- Constants (both `constexpr`s and macros) should be in all-caps
- Exceptions can be made in a variable's name when it would damage code readability or mathematical coherence
- Exceptions to `camelCase` can be made if the variable has a single-letter word. ex: `x`, `y`, `n`

27.4 Reconstruction code basis

- Core coding language: C++ with c++20 standard
  - Interface for Python
- Build system: `cmake`
- Testing tools:
  - C++: `catch2`
  - Python: `pytest`
- Multi-threading: `std::thread`
- GPU: `CUDA`

27.5 Priority functionality

- Scatter estimation with Time-of-flight
- Additive corrections
- GPU projector
- Motion correction directly from List-mode files(s)
- Multiplicative corrections
- PSF inclusion in the projector

27.6 Wish list for a full product

- **Fully 3D Multiple bed**
- Quantitative accuracy
- Dead time correction
- Decay correction
- Dynamic reconstruction
- Gated reconstruction

CHAPTER

TWENTYEIGHT

CURRENT YRT-PET VERSION:

```
2.0.4
```

CHAPTER TWENTYNINE

CITING YRT-PET:

If you use YRT-PET in one of your projects, you can cite our journal article in *IEEE Transactions on Radiation and Plasma Medical Sciences*.

```
@article{najmaoui2025,
    title = {{YRT}-{PET}: An Open-Source {GPU}-Accelerated Image
        Reconstruction Engine for Positron Emission Tomography},
    issn = {2469-7303},
    url = {https://ieeexplore.ieee.org/document/11202639},
    doi = {10.1109/TRPMS.2025.3619872},
    journal = {{IEEE} Transactions on Radiation and Plasma Medical
        Sciences},
    pages = {1--1},
    author = {Najmaoui, Yassir and Chemli, Yanis and Toussaint, Maxime and
        Petibon, Yoann and Marty, Baptiste and Fontaine, Kathryn and
        Gallezot, Jean-Dominique and Razdevšek, Gašper and Orehar, Matic and
        Dhaynaut, Maeva and Guehl, Nicolas and Dolenec, Rok and
        Pestotnik, Rok and Johnson, Keith and Ouyang, Jinsong and
        Normandin, Marc and Tétrault, Marc-André and Lecomte, Roger and
        El Fakhri, Georges and Marin, Thibault},
    year = {2025},
    keywords={Image reconstruction;Biomedical imaging;Graphics processing
        units;Software;Positron emission tomography;Python;Engines;Software
        algorithms;Plasmas;Geometry;positron emission tomography (PET);image
        reconstruction;software;graphics processing unit (GPU);C++;Python}
}
```